MASTER EN SCIENCES INFORMATIQUES

Présentation du langage C

Perquy, Filip

Award date:
1988

Awarding institution:
Universite de Namur

FACULTÉS
UNIVERSITAIRES
N. D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

Année académique
1987 - 1988

Présentation du langage C

Filip Perquy

**Mémoire présenté en vue de
l'obtention du titre de
Licencié et Maître en Informatique**

Promoteur : Baudouin Le Charlier

Adresse : rue Grandgagnage, 21, B-5000 NAMUR (Belgium) - ☎ 081 - 22 90 65 - Télex 59.222 FAC NAM-B

RESUME

Aujourd'hui, le langage C est un langage de plus en plus utilisé. Il a été développé en même temps que le système d'exploitation UNIX. Ce mémoire présente une étude globale du C.

Les diverses constructions syntaxiques sont présentées. Ensuite, la sémantique du langage C est expliquée à l'aide de principes dénotationnels.

Ensuite, les temps d'exécution de plusieurs instructions individuelles et de programmes complets sont mesurés et interprétés.

ABSTRACT

The C langage has been developped at the same time as the operating system UNIX.

This thesis presents a global study of the C-language.

First of all, its syntactic constructs are explained.

The explication of the semantics of the C-language is based on denotationnal principles.

Some tests considering the execution time of some instructions and programs are also performed.

Je tiens à remercier :

monsieur Baudoin Le Charlier, pour les conseils apportés au cours de la réalisation de ce travail;

monsieur Bernard Thiry, de sa grande disponibilité et de son aide précieuse pour la rédaction de ce mémoire;

monsieur Philippe Lambert, pour ses remarques et propos pertinents énoncés lors de l'élaboration de ce mémoire;

toutes les personnes de l'Institut d'Informatique, ou extérieures à la faculté, qui ont permis de mener à bien cette étude.

Table de contenu

Liste des figures, tables et exemples

2.2 Les Exemples

2.3 Les Tables

3 Chapitre III

3.1 Les Figures

4 Chapitre IV

4.1 Les Figures

4.2 Les Tables

Introduction

Parmi les langages algorithmiques, le langage C connaît une grande diffusion et commence à être largement utilisé. On prête à ce langage des grandes qualités de flexibilité et d'efficacité, malgré une syntaxe relativement complexe. Mais ce langage correspond-il bien à toutes les espérances prétendues ?

L'objectif de ce mémoire est de répondre aux questions posées et ainsi de donner les points importants nécessaires à une étude efficace et rigoureuse.

Le premier chapitre : "Une typologie des langages de programmation" pose les principes primitifs afin d'établir une typologie des langages de programmation.

Sur cette base, le chapitre II : "Le langage C et son modèle de calcul" envisage une présentation du langage C.

Ensuite, un modèle sémantique permettant de correctement interpréter l'exécution d'une instruction est construit dans le chapitre III : "La sémantique du langage C".

Enfin, dans le dernier chapitre : "Des aspects de la performance du langage C" certains aspects de l'efficacité et des performances du langage C sont testés.

Chapitre I :

Une typologie des langages de programmation

1 Introduction

Il existe une grande variété de langages. Le Fortran est conçu pour les applications numériques, le Lisp et le Prolog sont orientés programmation symbolique, le C est très utilisé dans le domaine de la programmation système.

A première vue tous ces langages sont différents et servent à des buts différents. Mais n'existe-t-il pas des critères qui permettent de regrouper plusieurs langages qui seraient différents mais qui réagiraient de la même façon vis-à-vis de ces critères ?
On pense ici aux mécanismes mettant à jour la mémoire, ceux qui définissent et appellent des procédures ou des fonctions et ceux qui contrôlent l'exécution d'un programme.

Le but de ce chapitre est de dégager un ensemble de critères et de concepts qui permettent d'établir une classification des langages de programmation et qui, plus loin dans l'étude, serviront comme base pour la définition de la sémantique d'un langage.

La classification présentée dans ce chapitre, constitue une synthèse des idées présentées par J. Backus dans l'article "*Can Programming be liberated from the Von Neumann Style ? A Funtional Style and Its Algebra of Programs*" et par C. Strachey dans "*Towards a Formal Semantics*" (dans Formal Languages Description Languages for Computer Programming).

2 Qu'est-ce qu'un langage de programmation ?

Un ordinateur est un instrument qui doit aider les utilisateurs à résoudre leurs problèmes.

Cette machine ne sait pas fonctionner par elle-même, ses différents composants matériels doivent être gérés et contrôlés par un ensemble de programmes ou le **software**. Ces programmes traduisent une certaine idée, un certain agencement de la "structure" d'une solution informatique : l'algorithme.

Un algorithme décrit la séquence des opérations à effectuer sur un ensemble de valeurs en entrée, pour produire un ensemble de valeurs en sortie, qui forment le résultat de l'algorithme [MARCOTTY & LEDGARD p. 6].

"Un langage de programmation est un ensemble de conventions de notations pour décrire et communiquer des algorithmes" [TENNENT p. 1].

Traditionnellement, l'étude des langages de programmation se divise en la **syntaxe** et en la **sémantique** d'un langage.

La syntaxe concerne les éléments de la structure, de la forme des constructions dans un langage [TENNENT p. 2]. Il s'agit d'un ensemble de conventions et de notations formelles relativement mathématiques.

D'un point de vue syntaxique, un programme est représenté par un texte. Ce texte est composé de chaînes de caractères dont les symboles appartiennent à un alphabet prédéfini.

Pour la plupart des langages de programmation, cet alphabet est composé de 54 lettres (de "a" jusque "z" et de "A" jusque "Z") complété des dix chiffres et des signes de ponctuation comme le guillemet ("), la virgule (,), le point-virgule (;), le point (.), les accolades { }, les crochets [ ], etc. Il existe des exceptions comme APL qui n'est pas basé sur cet alphabet mais qui emploie un ensemble de signes pour exprimer les programmes.

La sémantique décrit le sens et la signification des constructions syntaxiquement correctes dans un langage.

Un petit aperçu historique

Du temps des premiers ordinateurs, dans les années cinquantes, les langages de programmation n'existaient pas. On programmait les opérations à effectuer en manipulant des interrupteurs, puis directement en code machine.

L'introduction des assembleurs amena une première amélioration. Ces langages permirent d'employer des noms symboliques pour les codes opératoires et les adresses.

Le Fortran fut le premier langage de programmation mis au point pour réduire le coût de développement d'un programme. Il permit la traduction automatique de formules mathématiques en instruction machine. Le Fortran a introduit la notion de modularité en employant des sous-programmes qui sont compilés séparément et qui peuvent accéder à un environnement global de variables. L'efficacité est un but très important dans le développement de ce langage [GHEZZI & JAZAYERI p. 13].

En 1959, apparaissait un autre langage qui est encore beaucoup employé aujourd'hui : le Cobol. Il était destiné pour un environnement d'entreprises où on doit traiter de grandes quantités de données. Le Cobol introduisait la notion de fichiers et la description de données [MARCOTTY & LEDGARD p. 4].

Algol 60 était le premier langage avec la notion de structure de bloc et de procédures récursives [GHEZZI & JAZAYERI p. 13].

Le Lisp et APL ont été développés dans une direction tout à fait différente. Ils étaient basés sur des théories mathématiques et n'étaient pas concernés par des considération d'efficacité [HOARE, p. 334].

A la fin des années '60, les langages Algol 68 et Pascal étaient mis au point. Le Pascal a eu le plus de succès. Conçu comme méthodologie pour apprendre à programmer, il est employé sur les micro-ordinateurs [JENSEN & WIRTH p. 153].

Aujourd'hui, les projets de recherche sont orientés vers la programmation logique avec le Prolog, fonctionnelle avec le Lisp et orientée objet avec Smalltalk.

3 Quels concepts se trouvent à la base d'un langage ?

3.1 Introduction

La section précédente offre une description des langages de programmation, de même que certains aspects historiques. Ce bref aperçu indique que ces systèmes de notations ne sont pas établis une fois pour toute mais évoluent grâce à de nouveaux développements dans la théorie et la pratique.

Dans cette partie de ce chapitre, on n'examinera pas tellement les causes historiques qui ont provoqué et stimulé les développements en matière de langages de programmation, mais on cherchera plutôt un ensemble de critères qui permettront d'établir une classification des différents langages.

D'abord, nous établirons une base sur laquelle une typologie pourra être construite, est établie. Ensuite, un modèle particulier avec ses caractéristiques sera étudié de plus près. Les constructions syntaxiques des langages appartenant à ce modèle seront réparties en plusieurs classes. Ceci offrira un outil pour déterminer si un langage est basé sur ce modèle ou pas.

3.2 Le langage et son modèle de calcul

Cette section établira les bases sur laquelle une typologie des langages peut être construite.

D'abord, certaines caractéristiques des langages de programmation seront considérées en toute généralité. Celles-ci permettront d'identifier deux classes dans lesquelles on pourra répartir l'ensemble de toutes les constructions syntaxiques et sémantiques d'un langage.

Ensuite, les mécanismes par lesquels le concepteur d'un langage s'est laissé guider pour déterminer le contenu de ces deux classes, seront étudiés.

Le schéma 1.1 représente ces deux idées.

3.2.1 **Les aspects déclaratifs et impératifs d'un langage**

Les symboles d'un alphabet, les lettres et les signes de ponctuation constituent les briques de base de la syntaxe. Ils sont combinés pour former des constructions syntaxiques de différents types comme des déclarations, des expressions ou des commandes. Quelques exemples peuvent illustrer cela. Pour le moment, on ignore comment ces différents types d'instructions sont identifiés ou à quelle sorte de langage qu'ils appartiennent.
Par exemple :

en Pascal,
```
var a : array [boolean] of integer;
```

en C,
```
int a[2];
```

en Prolog,
```
complet(X,Y) :- morceau1(X), morceau2(Y).
```

en Scheme (un dialecte de Lisp),
```
(define size 2)
```

Exemple 1.1 : une déclaration

en Pascal,
```
i := 0;
while (i =< 1) do
     Begin
          a[false] := i;
          a[true] := i; := i + 1;
     End
```

en C,
```
i = 0;
while (i =< 1) {
     a[i++] =
     }
```

Exemple 1.2 : une structure de contrôle

Ces exemples, basés sur Pascal, C, Lisp et Prolog montrent deux aspects fondamentaux des langages de programmation **en général**. Dans un langage, on peut distinguer d'un côté des constructions syntaxiques **prédéfinies** (appartenant donc à ce système de notations) et des constructions formelles définies **par l'utilisateur**.
Dans l'exemple 1.2 "while", "Begin End", "{ }", ":-", "var" et "array _ of _" font partie de ce cadre prédéfini. Par contre "a", "i", "var a : array [boolean] of integer", "(define size 2)" et les deux boucles de contrôle sont définis par l'utilisateur [BACKUS p. 617].

Ce **cadre prédéfini** contient toutes les caractéristiques syntaxiques et sémantiques d'un langage. Cette structure générale est définie à la conception et fournit un environnement général que l'utilisateur devra employer lors de la conception de ces programmes.

Ces constructions prédéfinies peuvent être réparties en deux groupes : les éléments **déclaratifs** et les éléments **impératifs**.

D'un côté, on peut regrouper toutes les instructions déclaratives d'un langage comme les différents mécanismes de déclaration de variables, de définitions de procédures, de fonctions de règles ou de faits. Lors de l'exécution d'un programme, ces éléments ne font rien. Tant ne qu'ils sont appelés, ils restent **passifs**.

Par contre, les constructions impératives **agissent**, elles provoquent l'exécution du programme et ont comme but de guider et de contrôler son déroulement. C'est le domaine des instructions séquentielles, conditionnelles, répétitives ou récursives, des instructions de branchement, des mécanismes d'affectation, d'unification, d'instanciation et de désinstanciation de variables.

3.2.2 Le modèle de calcul

Si pour un certain langage, on dispose de la répartition en constructions déclaratives et impératives, alors on peut se demander quels mécanismes et quels concepts de base ont déterminé le contenu de ces deux groupes ?

L'origine de la syntaxe et la sémantique de chaque langage est basée sur un certain nombre de principes théoriques et pratiques qu'on regroupera sous le concept "**modèle de calcul**". Quand un concepteur construit son système de notations formelles, il a un certain modèle de calcul en tête qu'il veut implémenter avec son langage.

Il en existe différents types. Un modèle peut être entièrement théorique ou il peut être inspiré par la technologie et des raisons pragmatiques.

Prolog, par exemple est basé sur la logique mathématique du calcul des prédicats de premier ordre et des clauses de Horn. Cette base théorique a donné lieu à un moteur d'inférence, qui accomplit le côté impératif de Prolog. La définition des règles et des faits est l'aspect déclaratif de ce langage. Prolog et les autres langages basés sur ce modèle sont appelés "des langages logiques".

Le Lisp est un autre exemple. Ce langage est basé sur les fonctions récursives et le calcul symbolique [SAMMET p. 405].

Le modèle qui a servi comme base pour la conception des langages comme Fortran, Algol, Pascal, Ada, etc ... est un modèle **opérationnel** basé sur le principe d'**abstraction** et influencé par des considérations du **développement de logiciels**.
Le modèle opérationnel se trouve aussi à la base du langage C. Comme dans le cas des autres langages, le C dispose d'instructions impératives et déclaratives inspirées par le principe d'abstraction et la théorie du développement de logiciels.

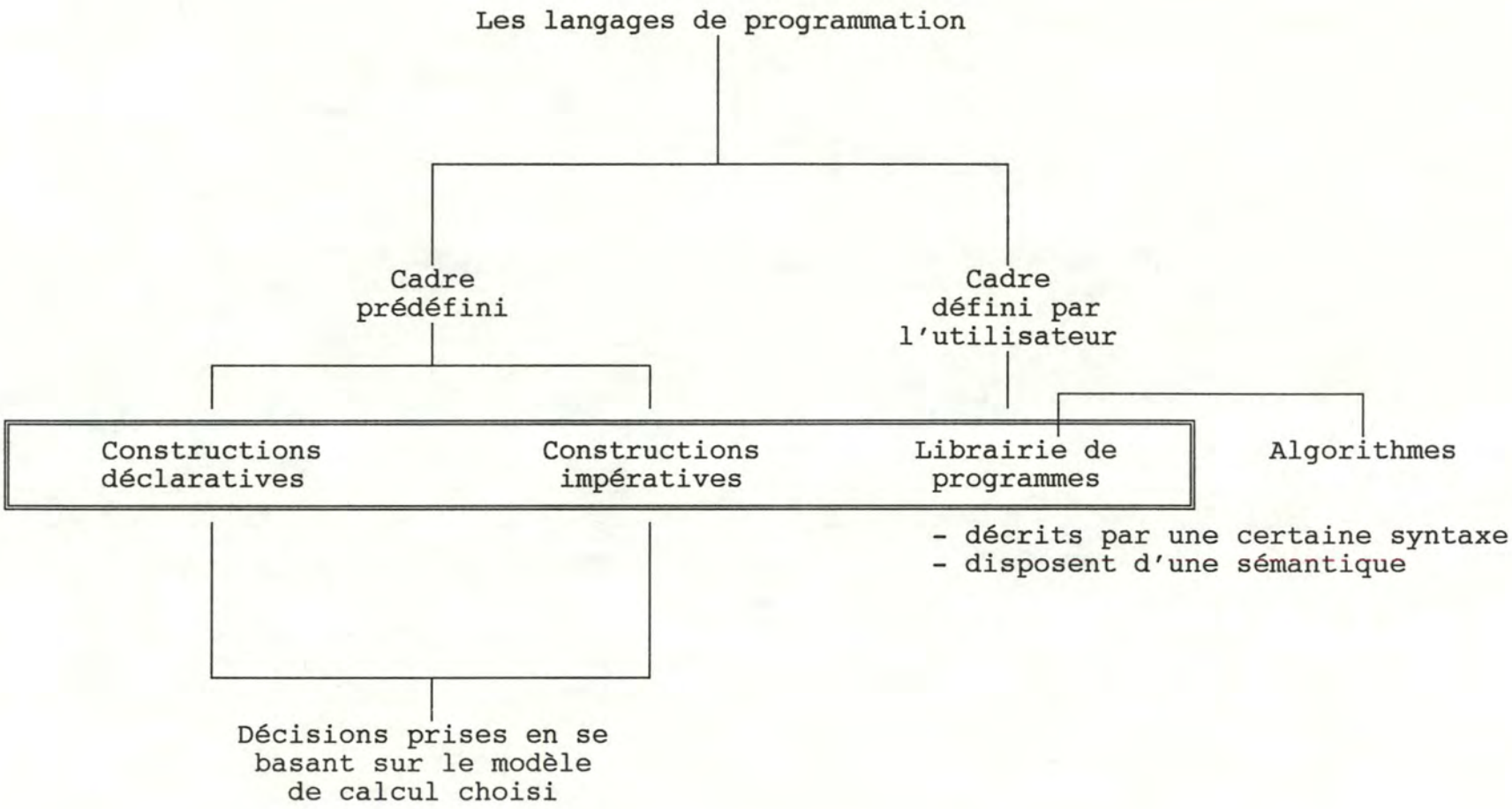


Figure 1.1 : Figure représentant les aspects déclaratifs et impératifs d'un langage

3.3 Les briques de base des langages conventionnels

3.3.1 L'architecture Von Neumann avec le mécanisme d'abstraction

Une des bases du système de calcul pour les langages de programmation est le modèle du hardware des ordinateurs conventionnels. La conception de ces langages est fortement influencée par le fonctionnement du hardware d'un ordinateur basé sur l'architecture **Von Neumann**. Dans leur forme la plus simple, les machines ayant une telle architecture sont composées des éléments suivants : un processeur central (C.P.U. ou central processing unit), une mémoire et un bus.

Le C.P.U. gère l'exécution d'un programme. La mémoire permet de stocker des informations et des instructions . Le bus accomplit le transfert des données et des opérations entre la mémoire et le processeur. Le schéma suivant représente cette architecture classique.

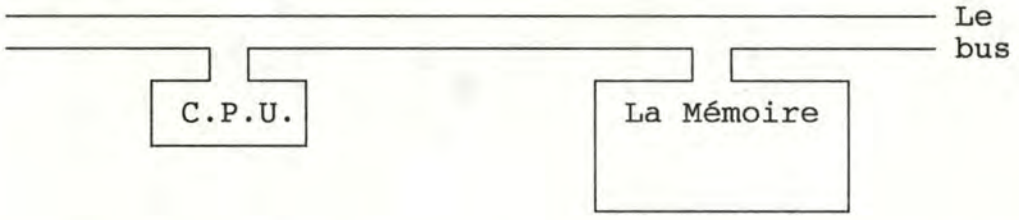


Figure 1.2. : représentation d'une architecture "Von Neumann"

Par la suite, les systèmes de notations construits selon ce modèle seront appelés les langages conventionnels ou Von Neumann.

Les premiers ordinateurs étaient directement programmés en code machine. Le programmeur devait donc gérer lui-même tous les détails d'adressage nécessaires à l'exécution de son programme.

Le mécanisme d'abstraction a permis de porter la programmation à un niveau plus haut. L'abstraction est un procédé permettant de considérer à part un élément d'une représentation ou d'une notion, en portant spécialement l'attention sur lui et en négligeant les autres.

Les assembleurs constituent une application de ce principe. Ils permettent au programmeur de travailler avec des instructions mnémoniques remplaçant les codes opératoires du langage machine et acceptant des noms symboliques au lieu des adresses absolues [MEINADIER Jean-Pierre, p. 53].
Les assembleurs ou plus tard dans l'évolution historique, les compilateurs et les interpréteurs prennent soin de ces aspects de la création d'un programme exécutable.

Ce procédé permet donc de cacher la structure matérielle de l'ordinateur sur lequel le programme va s'exécuter, ceci rendant la programmation plus conviviale vis-à-vis de l'utilisateur-programmeur. Il a l'impression de travailler sur une machine abstraite.

L'architecture Von Neumann et le procédé d'abstraction ont exercé une première influence déterminante sur le système de calcul à la base des langages conventionnels.

3.3.2 L'influence exercée par le développement des logiciels

Le second pilier à la base du modèle de calcul des langages conventionnels constitue la théorie du développement de logiciels et associé à celui-ci, les considérations théoriques sur la construction de programmes.

Dans le cycle de vie d'un projet informatique, on peut distinguer plusieurs phases. Traditionnellement, on commence avec l'analyse des besoins et l'analyse fonctionnelle, suivi par la conception d'une architecture logique et plus tard par une architecture physique. Lors de cette phase seulement, interviennent les langages de programmation comme outils qui doivent aider à implémenter la solution abstraite sur une machine concrète [VAN LAMSWEERDE p. 30].

Pendant la phase du développement d'une architecture logique, on construit un certain nombre de concepts et d'idées, qui seront employés dans la solution informatisée. Il s'agit par exemple, d'une automatisation d'un service de vente où on a décidé de représenter l'entité "client" par son nom, son adresse, le total d'achats qu'il a déjà effectués, la date du dernier achat, etc.

Lors de la programmation, ces concepts doivent être exprimés à l'aide du langage choisi. Supposons que le langage dispose d'un mécanisme de structuration "record", qui permet de composer des nouveaux types à partir de types de base comme les entiers, les réels et les chaînes de caractères. En employant ce "record", on peut implémenter l'entité "client", sans trop de problèmes. Le programme reflète ainsi les concepts identifiés par l'architecture logique.

La construction de la structure "client" est aussi basée sur le mécanisme d'abstraction (déjà invoqué lors de la discussion sur les liens entre le hardware d'un ordinateur et les concepts d'un langage). Dans ce cas-ci, il s'agit d'une abstraction construite par le programmeur. Le type "client" n'existe pas en tant que tel dans le langage, il appartient aux constructions définies par le programmeur (cfr. 3.2.1 où on sépare les aspects prédéfinis d'un langage de ceux conçus par l'utilisateur).

Seulement, la structure ("record") qui a permis d'implémenter "client", appartient au système de notations choisi et la décision de fournir ce type est influencée par le processus de développement de logiciels et l'évolution de la théorie.

Il existe d'autres exemples de cette interaction entre la théorie de la conception des logiciels et les langages de programmation.

Le Pascal par exemple, a été conçu pour supporter la programmation structurée et la construction d'un logiciel dans le style "top-down" en éliminant le plus possible les instructions de branchement [JENSEN & WIRTH p. 153].

Le "package" en Ada, pour mentionner un autre exemple, permet d'appliquer le concept de module. Chaque module cache un certain nombre de secrets, qui sont souvent des structures de données. Pour permettre aux autres modules de consulter ou mettre à jour l'information contenue dans cette structure, ce module leur offre des fonctions d'accès sans qu'ils ne puissent directement manipuler cette structure et sans connaître la façon dont sont implémentées ces fonctions. Ces primitives forment une application du principe de rétention de l'information. Elles permettent de concevoir des logiciels dont la maintenance et les modifications sont plus faciles [GHEZZI & JAZAYERI, p. 5].

3.4 Les classes syntaxiques des langages conventionnels

A la section 3.2.1, deux classes d'instructions ont été identifiées. Elles permettent de répartir les constructions syntaxiques de tout langage en instructions déclaratives et impératives.

Cette répartition doit être affinée pour les langages conventionnels.
On distingue dans la classe déclarative :

- les déclarations et définitions avec les structures de bloc,
- les expressions,
- les procédures et les fonctions;

et dans la classe impérative :

- les commandes,
- les instructions de branchement.

Les langages de programmation disposent de types de données simples, par exemple : les entiers, les réels et les chaînes de caractères. Ces types de données de base constituent une abstraction du modèle de la mémoire. Une donnée est représentée par une configuration de bits stockés dans une ou plusieurs cellules adressables en mémoire. Par exemple, le programmeur peut directement manipuler l'entier '25' et n'a pas besoin de connaître la configuration de bits sous-jacente.

Ces types de base peuvent être agrégés en types de données composés définis par l'utilisateur. Plusieurs agrégats peuvent à nouveau être combinés pour représenter des données. On pense ici aux mécanismes de déclaration de tableaux, d'enregistrements (records), d'ensembles, etc ... .

Les données simples et composées font partie du programme. Le mécanisme de déclaration et de définition permettent de donner un nom à ces constructions. Le programmeur peut définir un nom symbolique pour une valeur constante ou variable. Ensuite, les données sont manipulables dans le programme par le seul emploi de ce nom. Après sa définition, un nom peut exister pour toute la durée d'exécution du programme ou peut être lié à un morceau de programme et avoir une durée de vie limitée. A ce moment, on parle d'une structure de bloc. La donnée est créée au début du bloc et détruite à la fin [GHEZZI & JAZAYERI p. 97].

Les données d'un programme peuvent être groupées à l'aide d'opérateurs pour former des expressions. Les expressions sont des mécanismes de calcul, qui permettent de produire une valeur à partir d'un ensemble d'autres valeurs en les combinant à l'aide d'opérateurs. Par exemple, (x + y) / (z - t) est une expression : (), +, /, - sont les opérateurs qui permettent de combiner les valeurs des variables x, y, z et t pour produire la valeur-résultat.

Les opérateurs d'adressage forment une autre application de ce mécanisme de calcul. L'adresse de la cellule mémoire indiquée par a[i], est calculée de la façon suivante : a est considéré comme l'adresse du début du tableau, i désigne le déplacement par rapport au début de ce tableau et "[ ]" est un opérateur d'adressage indiquant au compilateur qu'il s'agit d'un calcul d'adresse. Il dépendra du contexte de a[i] pour déterminer s'il faut stocker une valeur dans la cellule indiquée par cette adresse ou s'il faut extraire la valeur qui y est enregistrée.

Les procédés de définition de procédures et de fonctions sont des mécanismes d'abstraction plus évolués que ceux des commandes et des instructions de branchement. Ils permettent au programmeur de séparer les niveaux d'abstraction dans le développement de grands programmes. Ces concepts rendent la programmation plus modulaire avec les avantages de modifiabilité et réutilisabilité qui en découlent [TENNENT p. 19].

Les ordinateurs conventionnels disposent de la propriété que le contenu de tout emplacement en mémoire peut être considéré comme l'instruction suivante à exécuter. Cette propriété constitue la base des classes impératives dans les langages conventionnels. Elle est utilisée pour implémenter des structures de contrôle qui permettent de diriger l'exécution d'un programme.

Les instructions de branchement sont des instructions de branchement vers un endroit particulier d'un programme. Il s'agit de branchements inconditionnels et directs. L'origine de cette classe peut être directement retrouvée dans les instructions de contrôle du "program counter". Le "program counter" constitue une partie du processeur central d'un ordinateur. Il contient l'adresse de l'instruction suivante à exécuter. L'exécution d'une instruction de branchement provoque le changement de l'adresse contenue dans cet indicateur d'instruction.

La classe des commandes couvrent le principe de l'assignation, de groupement des commandes ("begin" et "end" en Pascal et Algol), et de composition séquentielle, conditionnelle ("if") et répétitive ("while"). Les commandes sont des abstractions des instructions de contrôle en particulier et des instructions machine en général. La séquence exprime l'agencement des instructions les unes derrière les autres, comme le fait le "program counter". La sélection et la répétition présentent des abstractions de mécanismes offerts par le hardware pour modifier le contenu du "program counter" [GHEZZI & JAZAYERI p. 181].

La boucle, structure de composition séquentielle, par exemple, peut être décomposée en une instruction de test et de branchement conditionnel. L'assignation est une abstraction basée sur les opérations de "load" et de "store" d'une valeur à travers le bus. D'ailleurs, la composition séquentielle des commandes reflète justement les phases de chargement et d'exécution des instructions les unes après les autres.

Chapitre II :

Le langage C et son modèle de calcul

Jusqu'à présent, le langage C n'a presque pas été mentionné dans cette étude. Ceci est fait délibérément. On a voulu construire un cadre général, ce qui a permis de mieux comprendre la philosophie des langages de programmation.

Dans les sections précédentes, la notion de langage de programmation et celle de modèle de calcul ont été définies. On a étudié en détail les bases du système conventionnel. Ce modèle de calcul est fondé d'une part sur l'architecture d'un ordinateur et d'autre part sur la théorie du développement des logiciels.

Ensuite, on a réparti les constructions syntaxiques des langages conventionnels en déclarations et définitions de variables, en définition de procédures, en expressions, en commandes et en ruptures de branchement.

Dans cette section, un aperçu historique avec une description des classes syntaxiques du langage C sera présenté. Il existe beaucoup de références sur le C. La plus classique est certainement "*The C Language*" de Kernighan & Ritchie. Une autre référence importante est "*C : A Reference Manual*" de Harbinson et Steels, paru chez Pren"ice-Hall.

Plusieurs articles sur le C sont paru dans "SIGPLAN Notices", comme "*C : Towards A Concise Syntactic Description*" écrit par Fitzhorn et Johnson et "*Type syntax in the language C*" de Anderson.
Une série d'articles qui comparent le C, le Pascal et le ADA sont regroupés dans "*Comparing and Accessing Programming Languages : Ada, C and PASCAL*".

Les lecteurs intéressés peuvent trouver des informations historiques concernant le développement du langage C et UNIX dans le "Bell System Technical Journal" de Bell Laboratories (l'édition de juillet-août 1978, n° 6).

1 Historique du langage

Pendant les années 60, les discussions dans le monde de l'informatique se sont concentrées sur les langages de programmation (PL/1, APL, Simula, Algol 68, ...) et les systèmes d'exploitation (CTTS et Multics) [BOURNE J. p.1]. En Angleterre plus particulièrement, deux universités entreprennent le 'CPL' : the Combined Programming Language project, mais ce projet n'aboutit à aucun résultat pratique.

Historiquement le langage C, le système d'exploitation UNIX et le PDP-11 de DIGITAL EQUIPEMENT sont fort liés. On fixe habituellement le début de UNIX à 1968. En cette année, Ken Thompson met au point la première version de UNIX sur un PDP-7. En février 1971, UNIX devient opérationnel sur le PDP-11. En 1972, la deuxième version du système d'exploitation UNIX sort. Chacune de ces versions est encore écrite en assembleur.

A cette époque, Ken Thompson travaille sur le langage B. Ce langage et le BCPL (qui est un descendant immédiat du CPL) connaissent comme type de données que le mot machine (16 bits sur le PDP-11). Ceci rend l'accès à des bytes (8 bits) particulièrement difficile [RITCHIE D. M.p.5].

Ces deux langages sont à la base du C. Au début, le langage C ne contient pas de structures ni de variables globales. En 1973, ces deux constructions sont ajoutées et comme le langage est assez attractif, on commence à réécrire le système UNIX en C [BOURNE J. p.3]. La taille du nouveau système augmente d'un tiers par rapport à la version en assembleur, mais contient de nouvelles caractéristiques : la multi-programmation et le code réentrant.
Le système est maintenant beaucoup plus facile à comprendre et à tenir à jour [BSTJ p.1931].

En effet, le C a les caractéristiques suivantes :

- Il est simple à employer. Il dispose d'un ensemble d'instructions réduites avec une syntaxe très concise.
- C'est un langage proche du système d'exploitation en particulier du système UNIX qui a été écrit à 90 % en C. Par conséquent, toute programmation orientée système sous UNIX s'effectue de manière aisée en C (accès facile au noyau UNIX).
- Le lien avec le système d'exploitation permet d'accéder très facilement à un grand nombre de mécanismes et de fonctions spécifiques, par l'intermédiaire d'une bibliothèque standard. Toutes les opérations d'entrée/sortie, par exemple, sont définies de cette manière.
- C possède un jeu très riche d'opérateurs, ce qui permet l'accès à la quasi-totalité des ressources de la machine. On peut faire de l'adressage indirect via les pointeurs ou utiliser des opérateurs d'incrémentation ou de décalage. On peut aussi préciser qu'on souhaite stocker une variable dans un registre. En conséquence, on peut écrire des programmes presque aussi efficaces qu'en assembleur, tout en programmant de manière structurée.
- De façon analogue à Algol 68, la plupart des constructions en C (fonctions ou expressions) renvoient une valeur, ceci permet de s'exprimer de façon souvent plus concise que dans d'autres langages.
- Le langage est faiblement typé. Cette permissivité relativement grande permet l'utilisation judicieuse des ressources matérielles, mais peut aussi évidemment conduire à de mauvaises habitudes de programmation. C'est pourquoi un programme supplémentaire, lint, existe en tant qu'utilitaire propre, permettant un contrôle plus rigoureux de la sémantique et des correspondances de types, indiquant des incohérences non détectées par le compilateur. Il est à noter toutefois que le programmeur peut très bien décider de garder ces incohérences, ou même par exemple de jouer sur elles à des fins d'efficacité.

Le laxisme du langage et du compilateur n'est cependant pas sans danger. Il peut en effet aboutir à des erreurs d'exécution difficiles à déceler, par exemple un débordement de tableau non testé ou un point-virgule oublié ou mal placé, entraînant un comportement parfois déroutant. Il faut donc

programmer de manière précise et claire, d'autant plus que les programmes C ont une certaine tendance à être peu lisibles si on ne fait pas un effort particulier au niveau de la présentation et de la rigueur [TOMBE, Karl. p. 1].

2 Aspects de la syntaxe

Il existe plusieurs versions différentes du langage C. Le temps et les machines ont apporté de légères modifications au langage. Dans ce mémoire, on se basera sur les documents suivants : "*The C Programming Language - Reference Manual*" (Dennis M. Ritchie, Bell Laboratories, Murray Hill, N.J., 1978) et "*The C Programming Language*", version française (Kernighan, B. W. et Ritchie D. M., Masson, Paris, 1987). Les programmes donnés comme exemples ont été testés sur un VAX 750 opérant sous le système d'exploitation UNIX 4.2 BSD.

Les entrées/sorties ne font pas partie du langage C. Elles appartiennent à la bibliothèque standard et ne sont pas décrites dans le "Reference Manual". Comme ces aspects ont une grande valeur pratique, le C sera considéré dans son environnement avec les entrées/sorties. Ces entrées/sorties sont décrites dans "The C Programming Language".

Le langage C emploie un alphabet tout à fait conventionnel, basé sur 54 lettres (minuscules et majuscules), 10 chiffres et les signes de ponctuation classiques. Les identificateurs suivants ne peuvent être employés qu'en tant que mots clés :

int	*extern*	*else*
char	*register*	*for*
float	*typedef*	*do*
double	*static*	*while*
struct	*goto*	*switch*
union	*return*	*case*
long	*sizeof*	*default*
short	*break*	*entry*
unsigned	*continue*	*enum*
auto	*if*	

Table 2.1 : les mots clés dans le langage C

Cette énumération des mots clés est reprise du "Reference Manual" [REFERENCE MANUAL p. 1]. Le mot clé *enum* y est ajouté. La répartition de ces mots dans les classes "déclaration & définition", "expression", "procédures & fonctions", "commandes" et uruptures de séquence" est représenté par la figure 2.1 qui se trouve à la fin de cette section.

Le langage C est "case-sensitif", c'est-à-dire que *essai* et *ESSAI* sont traités comme deux identificateurs distincts. Les noms des variables sont par convention écrits en minuscules, ainsi on les distingue, des noms symboliques traités par le précompilateur qui eux sont écrits en majuscules [KERNIGHAN B. W. & RITCHIE D. M. p. 16].
Dans un programme source, les commentaires sont introduits avec les caractères /* et terminés avec */.

Un programme C se compose d'un ensemble de **modules** compilés séparément. Un synonyme pour module est fichier. Chaque module comporte un certain nombre de fonctions et éventuellement des déclarations de variables globales. Toutes ces fonctions sont déclarées au même niveau, l'imbrication de fonctions n'est pas possible.

Dans l'ensemble des modules, une fonction particulière, ayant pour nom **main** doit obligatoirement appartenir à un des modules, et de manière unique. C'est le programme principal. Il forme le point d'entrée pour l'exécution du programme [TOMBE, Karl. p. 10].

Les paramètres formels d'une fonction doivent être séparés par des virgules. Les fonctions ont la syntaxe suivante :

```
nom-fonction(liste de paramètres formels)
spécification des paramètres formels;
{
déclaration des variable locales

suite d'instructions
}
```

Un exemple classique est :

```
main()
{
     printf("hello, world\n");
}
```

[KERNIGHAN B. W. & RITCHIE D. M. p. 9].

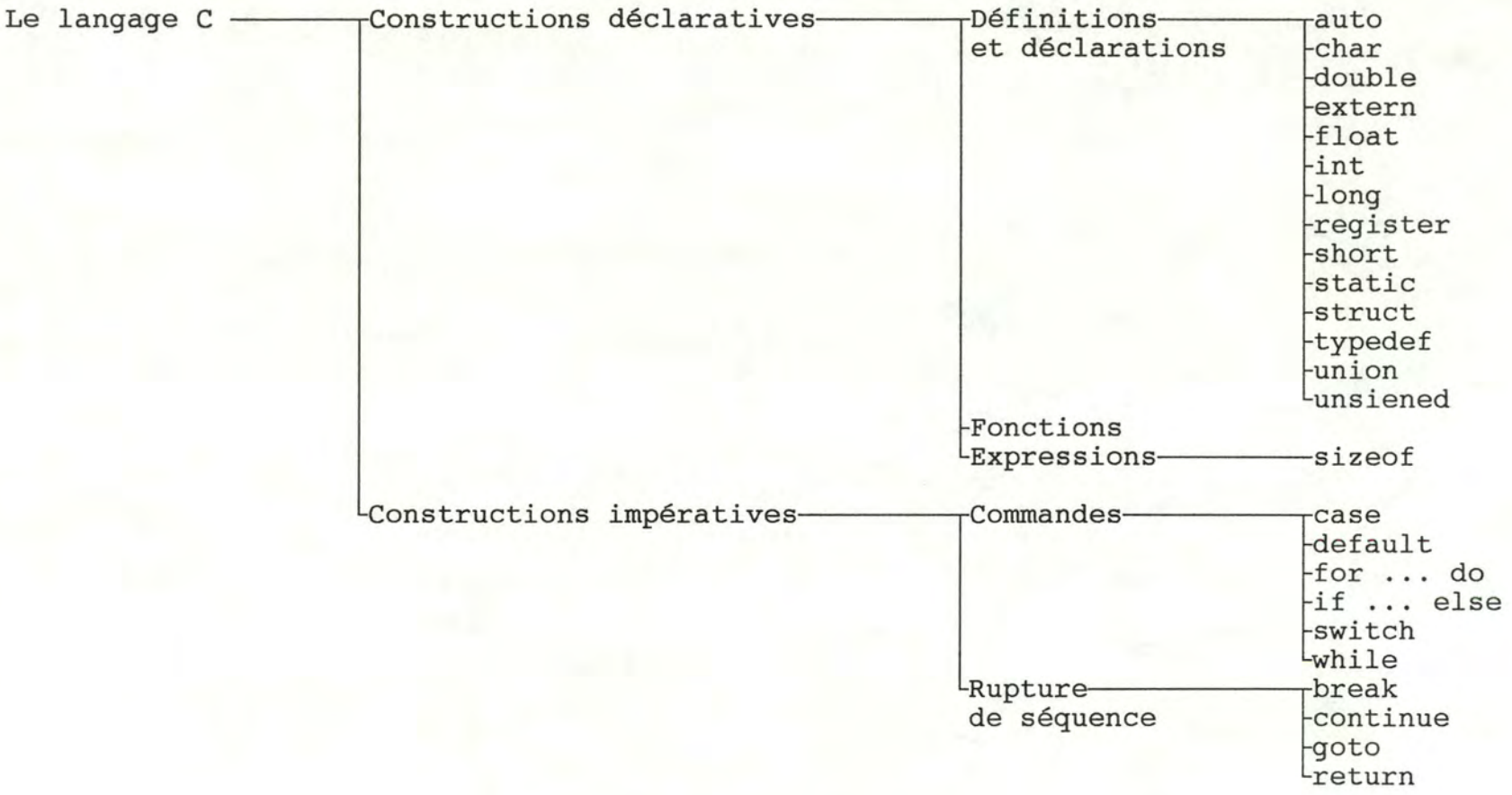


Figure 2.1 :
La répartition des mots clés dans les classes syntaxiques "Définitions et Déclarations", "Fonctions", "Expression", "Commandes" et "instructions de rupture de séquence"

3 Les classes syntaxiques en C

3.1 Les déclarations et les structures de blocs

3.1.1 Classe de mémorisation

La classe de mémorisation d'une variable détermine si elle a une durée de vie "globale" ou "locale".

Un objet ayant une durée de vie "globale", existe et possède une valeur dans le fichier source dans lequel il a été déclaré. Concrètement, ceci revient à dire que cet objet existera pendant toute l'exécution du programme [Reference Manual p. 3].

Dans le langage C, il est important de distinguer la **déclaration** d'une variable externe et sa **définition**. Une déclaration présente les propriétés d'une variable (type, taille, etc ...), une définition lui réserve une place en mémoire. Une variable externe ne doit être définie qu'une seule fois dans tous les fichiers qui constituent le programme source, les fichiers autres que celui où se trouve la définition peuvent comporter des déclarations *extern* pour y avoir accès [KERNIGHAN B. W. & RITCHIE D. M. p. 75].

Les variables avec une durée de vie "locale" reçoivent une nouvelle allocation mémoire à chaque fois que le bloc dans lequel elles sont définies, est exécuté.

Bien que le C ne connaisse que ces deux types de classes, on dispose de 4 spécificateurs suivants : *auto*, *extern*, *register* et *static*. Ils sont appelés "indicateurs de classe de mémorisation".

auto : c'est l'option par défaut pour toute variable interne d'une fonction. Ces variables ont une durée de vie locale et leur portée est limitée au bloc où elles sont définies.

extern : ce sont les variables définies à l'extérieur de toute fonction, et qui sont donc globales. Si on fait référence dans une fonction à une variable

définie dans un autre module (donc dans un autre fichier, et en compilation séparée), on précisera qu'elle est externe par le mot-clé *extern*.

static : les variables statiques peuvent être internes ou externes à un bloc. Une variable interne statique reste locale à la fonction, mais garde sa valeur d'un appel à l'autre (valeur rémanente). Une variable globale statique est une variable qui, bien que globale ne sera visible que dans le module ou le fichier où elle est déclarée. Cela permet de cacher des détails d'implémentation et de réduire les risques de double définition dans un logiciel de grande taille. Le mot-clé pour indiquer qu'une variable doit être de ce type est *static*.

register : on peut demander qu'une variable de type scalaire soit implantée en registre, qui permet d'augmenter la vitesse d'exécution. Attention : seule une variable automatique peut êtrt de type registre. De plus, le mot-clé *register* ne donne qu'une indication au compilateur, on ne garantit pas que la variable sera bien dans un registre [TOMBE, Karl p. 6].

Le tableau 2.2 résume les combinaisons admissibles des mots-clé. Les "X" se trouvant dans une même colonne indiquent que les mots-clé sur les lignes qui correspondent aux "X", peuvent être combinées. Le tableau indique qu'une seule combinaison n'est admissible : il s'agit de *auto* avec *register*

	auto	extern	static	register
auto	X			X
extern		X		
static			X	
register				X

Table 2.2 : représentation sous forme d'une table de décision des combinaisons des mots clés

Remarques :

1° Une fonction est toujours un objet externe car le langage C ne permet pas la définition de fonctions imbriquées l'une dans l'autre. Une fonction peut aussi être statique (elle est donc définie avec le mot clé *static*), alors elle ne sera visible que dans le fichier où elle est définie et ne peut donc pas être appelée en dehors de ce fichier.

2° Les variables (sauf les tableaux automatiques) peuvent être initialisées à leur définition. Les variables externes et statiques sont initialisées à la compilation, les variables automatiques et celles du type registre sont initialisées au début du bloc [KERNIGHAN B. W. & RITCHIE D. M. p. 81].

L'exemple suivant donne une illustration de l'emploi des variables externes. Il s'agit de deux fichiers. Dans le premier, on définit et initialise la variable globale *v* qui est appelée par une fonction dans le deuxième fichier, et la variable automatique i qui est un entier non signé. Dans ce dernier fichier, *v* est déclaré comme une variable externe. Remarquez dans *source1.c* que la fonction *f()* est déclarée comme *double*, indiquant le type du résultat d'un appel éventuel. Dans la déclaration, il n'y a pas d'information sur le nombre de paramètres.

Fichier "source1.c",

```
int v = 3;
main()
{
     unsigned i = 5;
     double f();
     printf("%d %d %f\n", i, v, f(v));
}
```

Fichier "source2.c"

```
double f(x)
double x;
{
     extern int v;  /* le compilateur va chercher v dans
                       ce fichier ou à l'extérieur */

     return(3.0 * x * v);
}
```

La fonction *printf("%d %d %f\n", i, v, f(v))* est une instruction d'entrée/sortie et appartient donc à la bibliothèque standard. Elle imprime sur la sortie standard trois valeurs (correspondant au *i*, *v*, et le résultat de *f(v)*) dans le format suivant "5 3 27".

3.1.2 Les types de base

Les types de base en C sont :

char	caractère (en général 8 bits)
int	entier (16 ou 32 bits, suivant les machines)
float	réel (32 bits, dépend de la machine)
double	réel en double précision (64 bits)
enum	énumération ou ensemble

Le type *int* peut être modifié par l'un des **qualificateurs** suivants :

short	entier court (souvent 16 bits),
unsigned	non signé,
long	entier long (souvent 32 bits).

La déclaration d'un tableau se fait de la manière suivante :

```
int a[10];
```

a est le nom d'un tableau de 10 entiers, indicé de 0 à 9.

3.1.3 La commande "Typedef"

La commande "typedef" permet de renommer un type. Elle définit principalement un synonyme pour des types existants. Les variables déclarées de cette manière possèdent exactement les mêmes propriétés que les variables déclarées sans le typedef [KERNIGHAN B. W. & RITCHIE D. M. p. 135].

Par exemple, la déclaration

```
typedef int LENGTH;
```

fait du nom *LENGTH* un synonyme de *int*. Le type "*LENGTH*" peut alors être utilisé dans des déclarations, comme par exemple dans

```
LENGTH    len, maxlen;
LENGTH    *length[];
```

La déclaration de *typedef char *STRING;* fait de *STRING* un synonyme de *char ** (qui est une variable qui pointe sur un caractère). Celui-ci peut alors être utilisé dans des déclarations comme par exemple :

```
STRING    p, alloc();
```

Le typedef ne réserve pas de place en mémoire. Il est une sorte "d'indicateur de classe de mémorisation" [KERNIGHAN B. W. & RITCHIE D. M. p. 187].

Cette commande peut être utilisée en combinaison avec la compilation séparée. Cette technique permet de regrouper dans un fichier tous les concepts définis lors de l'architecture logique d'un programme.

3.1.4 Les pointeurs et les tableaux

Les pointeurs sont des variables contenant des adresses. Ils sont utilisé sous la forme suivante : "*" est un opérateur unaire qui considérera la valeur de son opérande comme une adresse. Par exemple, **px* indiquera que la valeur contenue dans *px* doit être interprétée comme une adresse. La façon d'employer cette adresse dépendra du contexte. Par exemple, dans *y = *px;* il faut accéder à la valeur contenue à l'adresse indiquée par **px* et cette valeur doit être assignée à *y*. Dans **px = z;* par contre, le contenu de *z* doit être stocké à l'adresse indiquée par **px*.

Par conséquent, il y a deux notions à distinguer derrière la notation **px*. Cette différence se situe au niveau de l'emplacement en mémoire. Primo, il y a *px* qui est une variable et qui dispose d'un emplacement en mémoire. Le compilateur a associé le nom de "*px*" à cet emplacement. Secondo, il y a une autre cellule qui est anonyme, mais dont l'adresse est contenue à l'emplacement "*px*". La combinaison de * avec *px* permet d'utiliser cette cellule anonyme comme tout autre emplacement en mémoire portant un nom.

Le tableau 2.3 présente ce mécanisme schématiquement. Le symbole "α" représente l'adresse contenue dans *px*.
La variable pointée par *px* est notée **px*.

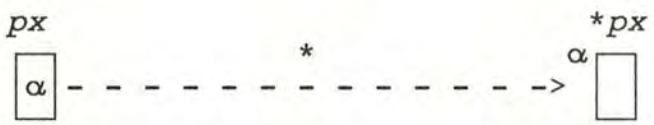


Tableau 2.3 : Représentation du mécanisme d'adressage avec un pointeur **px*

Comme toute autre variable en C, un pointeur doit être déclaré. La déclaration

```
int *px;
```

indique que la combinaison **px* est de type entier. Si *px* intervient sous la forme de **px*, elle équivalente à une variable de type entier.

Il est important de remarquer que :

1° la déclaration *int *px;* ne déclare qu'une seule variable : *px* qui est de type **pointeur sur un entier**,

2° *px* se comporte comme une variable tout à fait ordinaire, c'est-à-dire que après la déclaration, son contenu est indéfini et doit être initialisé avec une adresse, avant qu'on ne puisse l'employer dans un programme (cette initialisation peut se faire à l'aide d'un appel à l'allocateur de mémoire).

Inversement, pour une variable :

```
int x;
```

on peut accéder à l'adresse de *x* par la notation *&x*. Et ainsi, on peut écrire :

```
px = &x;
```

Cette instruction affecte l'adresse de *x* à *px*, par conséquent, *x* et **px* indiquent la valeur contenue dans la cellule mémoire *&x*.

Comme déjà mentionné, un tableau est déclaré de la manière suivante :

```
int a[i];
```

Il y a une très forte relation entre un pointeur et un tableau. Dans l'exemple *int a[i];*, *a* est en fait une **constante** de type adresse. En effet, *a* est l'adresse du début du tableau. Par conséquent, après la déclaration *int *pa, a[10];* on peut écrire les choses suivantes :

```
pa = &a[0];
ou                  /* ces deux affectations sont équivalentes */
pa = a;
```

Dans cet exemple, *&a[0]* calcule l'adresse du premier élément, ce qui est aussi bien indiqué par *a*, et on peut accéder aux éléments à l'aide de *a[i]* ou **(pa + i)*.

Néanmoins, il y a des différences dues au fait que *a* est une adresse (et donc une constante) alors que *pa* est une variable.
Ainsi, on peut écrire :

```
pa = a;
```

mais pas :

```
a = pa; /* sémantiquement impossible car s'il agit d'une
           affectation à une constante */.
```

En toute généralité, chaque fois qu'un identificateur de tableau apparaît dans une expression, le compilateur le convertit en une variable pointant sur le premier membre du tableau. A cause de cette conversion, les adresses des tableaux sont contenues dans des pointeurs. Par définition, l'opérateur d'indiçage *[]* est interprété de telle façon que $E_1[E_2]$ soit identique à $*((E_1) + (E_2))$, la somme des valeurs contenues entre les parenthèses est interprétée comme une adresse grâce à l'opérateur * [Reference Manual p. 23].

Quand on veut passer un tableau comme paramètre formel d'une fonction, il est équivalent d'écrire :

```
funct(tab)          ou          funct(tab)
int tab[];                      int *tab;
```

Quand le nom d'un tableau est passé à une fonction, l'adresse du début du tableau est transmise. A l'intérieur de la fonction appelée, cet argument est une variable ordinaire. Le compilateur interprète *tab[]* comme **tab*.

Le langage C permet de faire un certain nombre d'opérations sur les pointeurs.

Tout d'abord, on peut leur ajouter ou leur soustraire un entier n. Cela revient à ajouter à l'adresse courante *n* fois la taille d'un objet du type pointé. Ainsi, dans l'expression *pa + i*, avant d'être ajouté, *i* est multiplié par la taille des objets sur lesquels pointe *pa* [KERNIGHAN B. W. & RITCHIE D. M. p. 91].

On peut comparer deux pointeurs avec les opérateurs relationnels pour tester leur position relative dans un même tableau. On peut soustraire deux pointeurs. Naturellement le programmeur doit veiller qu'ils pointent dans la même zone mémoire

La figure 2.2 donne une illustration d'une comparaison de deux pointeurs. L'instruction accompagnante change un des deux pointeurs de position en fonction du résultat.

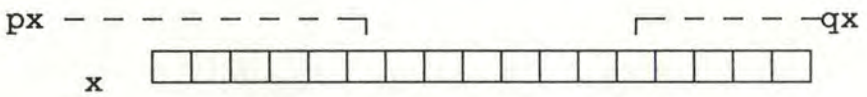


Schéma représentant la position relative de chaque pointeur dans le tableau

```
if (p < q )
        p++;
    else
        q++;
```

Figure 2.2 : Une comparaison de deux pointeurs

Des "affectations globale" sur des chaînes de caractères ou des tableaux où on copie la chaîne ou tableau entièrement, sont uniquement possibles par l'intermédiaire de pointeurs, comme illustré dans l'exemple suivant :

```
char *c;
c = "mémoire";
```

Lors de l'assignation, la variable *c* reçoit un pointeur vers "*mémoire*", *c* contient donc l'adresse du début de la zone où se trouve "*mémoire*".

La déclaration *char s;* est différente de *char *c;*. Il s'agit dans le premier cas (*char s;*), de la déclaration d'un seul **(1!)** caractère. Huit bits sont alors réservés. Dans le deuxième cas *char *c;*, **un pointeur vers un caractère (1!)** est créé.

En reprenant l'exemple *c = "mémoire";*, la construction *c[5]* renvoit le sixième charactère de la chaîne (c'est-à-dire le caractère *'r'*). L'instruction *p = *c++;* affecte le caractère sur lequel *c* pointe à *p*. Après l'affectation, le contenu de *c* est incrémenté de un, tel que **c* pointe alors sur le caractère suivant.

Les chaînes de caractères sont des tableaux de caractères terminés par le caractère nul (représenté par le caractère *'\0'* ou *0* est l'entier *0*). La construction *'r'* est différente de *"r"*, c'est-à-dire que *'r'* représente un seul caractère qui remplace la valeur numérique de la lettre *r* dans le code de la machine, le second est une chaîne de caractères ne comprenant qu'un caractère (la lettre *r* suivi du *'\0'*).

Les exemples précédents montrent que travailler avec des pointeurs n'est pas toujours chose facile. Après la création par le compilateur, les pointeurs (et d'ailleurs toutes les variables) contiennent des valeurs indéfinies.
L'initialisation des pointeurs avec des adresses valides est laissée à la responsabilité du programmeur. Dans les exemples suivants où des pointeurs sont employés, il est toujours supposé qu'ils sont correctement initialisés avec l'adresse des objets sur lesquels qu'ils pointent.

Remarque sur la syntaxe des déclarations

Comme la syntaxe des déclarations en C est basée sur la syntaxe des expressions, on peut employer des opérateurs dans des déclarations. Un idenficateur peut être entouré de parenthèses, qui permettent de construire des déclarations complexes.

Dans l'interprétation d'une déclaration complexe, les [] et les () ont priorité sur *. Les [] et () ont une même priorité et suivent la règle d'association de gauche vers la droite.

En toute généralité, la table 2.3 des priorités des opérateurs de la section 3.3.2 peut être repris pour lire des déclarations complexes. Pour raisons de facilité, il vaut mieux lire des déclarations complexes de l'intérieur vers l'extérieur.

Quelques exemples peuvent illustrer ceci.

*int *var[5]*	*var* est un tableau de 5 éléments qui sont des pointeurs vers des entiers
*int (*var)[5]*	*var* est un pointeur vers un tableau de 5 éléments de type entiers

[KERNIGHAN B. W. & RITCHIE D. M. p. 102].

Les deux constructions dans leur ensemble sont de type entier. Dans les deux cas, après la déclaration, uniquement une structure virtuelle est définie, les deux tableaux n'existent donc pas encore. Dans la deuxième exemple *var* est de type pointeur vers un élément d'un tableau qui doit recevoir une adresse.

La figure 2.3 représente schématiquement la structure virtuelle définie par des deux déclarations.

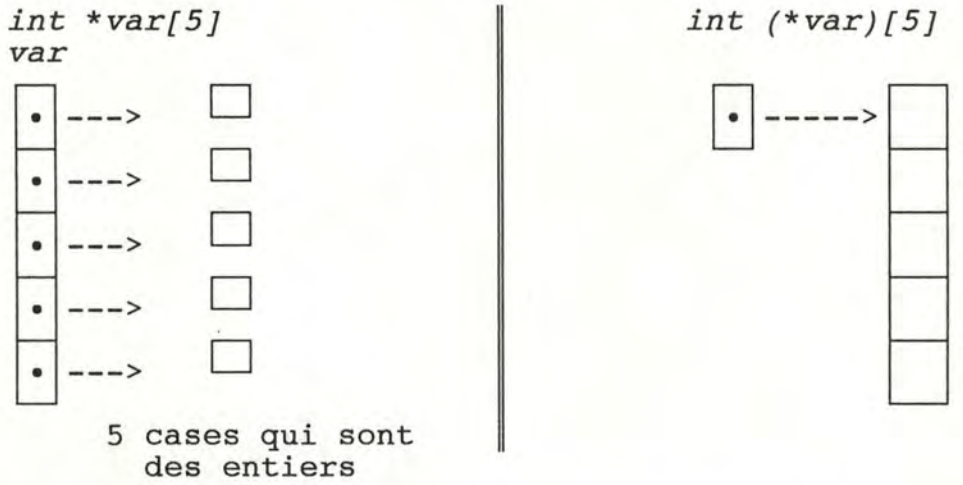


Figure 2.3 : représentation de deux structures virtuelles créées par des déclarations

3.1.5 Les structures et les unions

a. Les structures

Une structure est un objet comprenant une liste de membres qui sont les champs de la structure. Chaque membre peut être de n'importe quel type, il peut donc aussi être de type *struct* [RITCHIE, D. M., p. 12]. Les structures sont analogues au RECORD en PASCAL.

```
struct date {
     int day, month, year;
     char month_name[10];
     char day_of_week[4];
     } d1, d2, d3;
```

Dans cet exemple, on définit la structure ayant pour nom *date*, et en même temps, on déclare les trois variables *d1*, *d2*, *d3* qui ont le type *date*.

Le langage C offre la possibilité de définir la structure d'un côté et de déclarer les variables ailleurs. La définition d'une structure ne réserve pas de place en mémoire. Elle décrit un modèle ou la forme d'une structure. Dans l'exemple suivant, *date* sert de nom de type. Cet identificateur suffira plus tard pour définir des exemples réels de cette structure.

```
struct date {
     int day, month, year;
     ...
     };   /* fin de la définition de la structure */

struct date t1, d2, d3;
          /* déclaration des variables */
```

Les déclarations récursives d'une structure ne sont pas admises. Une structure peut toutefois avoir pour champs des pointeurs de structure quelconques. Cette technique permet d'implémenter des types récursifs tels que les listes chaînées, comme dans l'exemple suivant :

```
struct list {
     int donnée;
     struct list *suivant;
     } n;
```

*Struct list *suivant;* est la déclaration d'un pointeur vers une structure de type *list*. Avant l'emploi de ce pointeur, il faut qu'il soit initialisé avec l'adresse de l'élément suivant de la liste.

Les seules opérations possibles avec une structure sont le calcul de son adresse (avec &) ou l'accès à un de ces membres. Cela implique donc que l'on ne peut pas recopier ou affecter une structure en la considérant dans son ensemble, ni la transmettre à une fonction [KERNIGHAN B. W. & RITCHIE D. M. p. 117].

Pour accéder à un des champs d'une structure, on utilise l'opérateur "point", comme en PASCAL :

```
d1.day = 16; d1.month = 8; d1.year = 1988;
d1.month_name = "Août"; d1.day_of_week = "J";
```

La structure *date* est alors initialisée avec la date "jeudi, 16/08/1988).

On peut aussi utiliser des pointeurs de structures pour accéder aux champs. La déclaration *struct list *p* déclare un pointeur **p* qui pointe sur une structure du type *list*.

Il existe deux moyens pour accéder à une structure via des pointeurs. En combinant les opérateurs *, *()* et ., on peut accéder au champ *donnée*, comme c'est illustré dans la construction suivante *(*p).donnée*. La deuxième méthode consiste dans l'emploi de l'opérateur ->, *p->donnée* est alors équivalent et synonyme de *(*p).donnée*.

Dans *(*p).donnée*, les parenthèses sont nécessaires car . est plus prioritaire que * [KERNIGHAN, B. W. & RITCHIE, D. p. 118].

b. Les unions

Une "union" est une variable qui peut contenir (à différents moments) des objets de types et de tailles différents.

Les unions sont implantées de telle manière que les différents champs occupent la mémoire à partir de la même adresse, et par conséquent se recouvrent mutuellement [TOMBE, Karl. p. 18]. Les structures, par contre, sont physiquement implantées comme une juxtaposition des champs.

Sur le plan de la syntaxe, la déclaration d'une union est semblable à celle d'une structure. On accède aux membres d'une union en faisant "nom de l'union.membre" ou "pointeur union->membre" comme pour les structures.

3.1.6 **Les structures de bloc**

Une structure de bloc est un mécanisme syntaxique qui permet de regrouper plusieurs instructions et de les considérer comme une seule instruction. Par conséquence, les déclarations faites au début d'un bloc ont uniquement une signification dans ce bloc.

Chaque instruction est terminée par un point-virgule. Ce point-virgule indique la terminaison d'une instruction et non pas la séparation comme en PASCAL. Par exemple, en PASCAL, on écrit : "*f := f * i; i = i + 1*", en C par contre, on écrira : "*f = f * i; i = i + 1;*". Cette règle doit être appliquée, même si l'instruction se trouve à la fin d'un bloc.

En C, un **bloc** est une suite d'instructions délimitée par une accolade ouvrante "{" et une accolade fermante "}".
A l'intérieur de tout bloc, on peut aussi définir des variables locales à ce bloc.

```
test()
     {
     int n;
     ...
     if(n > 0) {
          int i;
          for(i=0; i < n; i++) {
                              ...
                              };
               };
     };
```

3.2 Les expressions

Les expressions forment un aspect important et riche du langage C. Le langage offre un jeu très étendu d'opérateurs avec 15 niveaux de priorité, ce qui permet l'écriture d'une grande variété d'expressions.

Dans cette section, l'influence des règles de priorité et d'association sur l'évaluation concrète d'une expression d'abord sera présentée. Ensuite, les opérateurs arithmétiques, relationnels, d'incrémentation et de décrémentation, logiques, conditionnels, d'affectation. Pour achever la section sur le langage C, les règles de conversion seront expliquées.

3.2.1 Généralités sur l'évaluation des expressions

Avant de pouvoir évaluer une expression, il faut disposer de valeurs en entrée et d'un certain ordre de calcul. Les valeurs en entrée résultent de l'effet dynamique de l'exécution du programme. L'ordre de calcul est partiellement basé sur un arbre de décomposition abstraite d'une expression.

L'arbre de décomposition abstraite est construit par le compilateur qui se base sur la priorité des opérateurs, la présence éventuelle de parenthèses et les règles d'association. Il utilise ces trois informations pour décomposer une expression en un arbre de sous-expressions. L'arbre détermine partiellement l'ordre d'évaluation. En se basant sur cet arbre, le compilateur détermine l'agencement complet de l'évaluation de l'expression. Par exemple, le compilateur peut arranger l'agencement de calcul des sous-expressions pour optimiser l'évaluation en temps d'exécution ou en espace mémoire.

La figure 2.4 illustre comment cet arbre de décomposition abstraite est constitué en se basant sur la priorité des opérateurs et les règles d'association.

Les expressions a*b + c, a * (b + c) et (a*b) + (c*d) ont les arborescences :

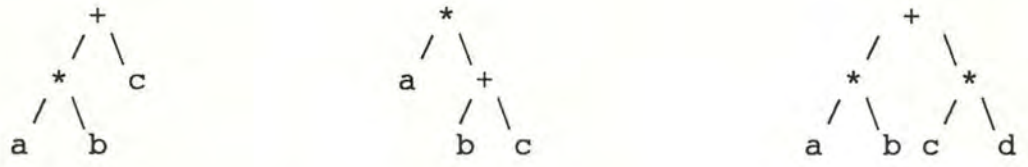


Figure 2.4 : Représentation sous forme d'arbre de a*b + c, a * (b + c) et (a*b) + (c*d)

Dans la troisième partie de la figure 2.4, l'évaluation par la machine peut éventuellement se faire en parallèle. Si l'exemple complet est a = (a*b) + (c*d), alors (c*d) sera probablement évalué avant (a*b), ainsi le compilateur peut garder l'adresse de a dans un registre pour faire l'assignation du résultat. Par contre, dans d = (a*b) + (c*d), l'évaluation des sous-expressions sera probablement effectuée dans l'ordre inverse.

3.2.2 L'ordre de l'évaluation des expressions en C

Comme mentionné dans la section précédente, la priorité des différents opérateurs avec les règles d'association déterminent un arbre de décomposition abstraite d'une expression. La table 2.4 résume les lois de priorité ainsi que l'associativité de tous les opérateurs. Les différentes entrées dans la table sont rangées par ordre de priorité décroissante.

Opérateur	Associativité
() [] -> .	de gauche à droite
!~ ++ -- - (type) * & sizeof	de droite à gauche
* / %	de gauche à droite
+ -	de gauche à droite
<< >>	de gauche à droite
< <= > >=	de gauche à droite
== !=	de gauche à droite
&	de gauche à droite
^	de gauche à droite
|	de gauche à droite
&&	de gauche à droite
||	de gauche à droite
? :	de droite à gauche
= += -= *= etc ...	de droite à gauche
,	de gauche à droite

Table 2.4 : les priorités et les règles d'associativité en C

[KERNIGHAN B. W. & RITCHIE D. M. p. 48]

En C, l'ordre d'évaluation d'une expression n'est pas déterminé. En particulier, le compilateur se réserve le droit de calculer des sous-expressions suivant l'ordre qu'il trouve le plus efficace même si ces sous-expressions influent sur d'autres expressions. L'ordre dans lequel les effets de bord apparaissent n'est pas précisé. Les expressions où intervient un opérateur associatif et commutatif (*, +, &, ¦ et ^) peuvent être remaniées arbitrairement, même à l'aide de parenthèses [KERNIGHAN B. W. & RITCHIE D. M. p. 178].

Il existe une exception pour les expressions relationnelles (basées sur les opérateurs suivants (>, >=, <, <=, ==, !=, !, &&, ¦¦). Le langage C **garantit** l'ordre d'évaluation de gauche à droite d'une expression contenant les opérateurs &&, ¦¦ [TOMBE, Karl p. 4]. Ces expressions sont donc évaluées de gauche à droite et l'évaluation s'arrête dès qu'on sait si l'expression est vraie ou fausse [KERNIGHAN B. W. & RITCHIE D. M. p. 39].

3.2.3 Opérateurs arithmétiques

+	addition
-	soustraction
*	multiplication
/	division (entière ou réelle)
%	modulo (sur les entiers)
> >= < <=	comparaisons
== !=	égalité et inégalité
!	négation (opérateur unaire)
&&	ET relationnel
¦¦	OU relationnel

3.2.4 Opérateurs relationnels

Il n'y a pas de type booléen en C, on utilise donc des entiers, et usuellement 0 représente faux et toute valeur non nulle représente vrai [MATETI, P. p. 40]. Cette règle peut présenter des problèmes comme dans l'exemple suivant : 1 < 2 < 3 et 3 < 2 < 1 sont tous les deux évalués à 1. Les règles d'association (de gauche à droite pour < ou >) provoquent que cette expression de relation est évaluée de gauche à droite, de sorte que 1 < 2 donne 1 (entier) et 1 < 3 donne 1. 3 < 2 donne 0 (entier) et 0 < 1 donne 1.

Comme déjà mentionné, le langage C garantit l'ordre d'évaluation d'une expression relationnelle (de gauche vers la droite). Cela est intéressant dans de nombreux cas, et évite l'écriture de tests plus complexes, parfois nécessaires dans d'autres langages comme le Pascal, mais implique aussi qu'il y a moins de possibilités d'optimisation.

L'exemple classique du *while ((i <= n) and (a[i] ≠ 0)* qui nécessite une réécriture dans le style *while ((i <= n) and (not found))* en Pascal, est correct en C. L'identificateur *found* est une variable de type booléen.

3.2.5 Opérateurs d'incrémentation et de décrementation

++	incrémentation
--	décrémentation

Ces opérateurs, qui ne peuvent être appliqués que sur les types scalaires, peuvent s'employer de deux manières : en principe, s'ils préfixent une variable, celle-ci sera incrémentée (ou décrémentée) avant utilisation dans le reste de l'expression. S'ils la postfixent, elle ne sera modifiée qu'après utilisation.

Ainsi :

```
a = 5; b= 6;
c = ++a - b;
```

donnera à *c* la valeur 0, alors que

```
a = 5; b= 6;
c = a++ - b;
```

lui donnera la valeur -1.

Cependant, il faut faire attention dans les expressions un peu complexes où on réutilise la même variable plusieurs fois. Dans le C, comme dans la plupart des langages, l'ordre dans lequel les opérandes d'un opérateur sont évalués, n'est pas précisé. L'expression peut donc avoir des résultats différents suivant la machine utilisée.

Ainsi, l'exemple suivant *x = f() + g();*, il se peut que *f* soit évalué avant *g* ou vice versa. Si jamais l'une des fonctions modifie la valeur d'une variable globale dont dépend le résultat de l'autre fonction, *x* peut alors dépendre de l'ordre dans lequel se fait l'évaluation.

3.2.6 Opérateurs logiques

Ce sont les opérateurs permettant d'effectuer des opérations au niveau des bits (masquages, etc ...).

`&` AND, exemple : *a & 0x000F* extrait les 4 bits de poids faible (En C, un chiffre préfixé par *0x* est en hexadécimal)

`|` OR, ainsi, *b = b | 0x100;* met à 1 le 9ième bit de *b*.

`^` XOR

`<<` SHIFT à gauche *a = b <<2;* met dans *a* la valeur de *b* où tous les bits ont été décalés de 2 positions vers la gauche. Les bits laissés libres sont remplis avec des zéros.

`<<` SHIFT à droite

`~` complément à 1 (opérateur unaire).

Pour << et >>, l'opérande de droite ne peut être négative et les 2 opérandes doivent être de type entier.

3.2.7 Expressions conditionnelles

Une expression conditionnelle est de la forme *expression1 ? expression2 : expression3* elle est évaluée de la manière suivante :

si expression1 alors expression2
sinon expression3

Ce type d'expression est pratique par exemple, pour calculer le maximum de deux nombres sans passer par une fonction,

```
z = (a > b) ? a : b;
```

Cette construction pourrait bien sûr s'exprimer avec une structure conditionnelle de la forme *if (...) ... else ...*, mais l'écriture sous forme d'expression conditionnelle est plus compacte.

3.2.8 Opérateurs d'affectation

= affectation

Il faut bien noter que le signe = est l'opérateur d'affectation, et non de comparaison, cela prête parfois à confusion, et entraîne des erreurs difficiles à discerner. A noter aussi que l'affectation est une expression comme une autre, c'est-à-dire qu'elle retourne une valeur. Il est donc possible d'écrire :

```
a = b = c+2;
```

ceci revenant à affecter à *b* le résultat de l'évaluation de *(c+2)*, puis à *a* le résultat de l'affectation *b = (c+2)*, c'est-à-dire la valeur qu'on a donnée à *b*. Remarquez l'ordre d'évaluation de la droite vers la gauche.

Il arrive très souvent qu'on calcule la nouvelle valeur d'une variable en fonction de son ancienne valeur. C fournit pour cela un jeu d'opérateurs combinés, de la forme <*variable*> <*op*>= <*expression*> où <*op*> est un opérateur. Une telle expression est équivalente à l'expression : <*variable*> = <*variable*> <*op*> <*expression*>. Les opérateurs qui suivent cette règle sont les suivants :

+=, -=, *=, /=, %=, <<=, >>=, &=, |= et ^=

3.2.9 Conversion de types

Si l'utilisateur désire changer le type du résultat retourné par une expression arithmétique, le C dispose de deux mécanismes : les conversions implicites et les conversions explicites.

Un certain nombre d'opérateurs peuvent, selon leurs opérandes, engendrer des conversions de la valeur d'un opérandes d'un type dans un autre. Les règles de conversion implicite peuvent être résumées comme suit :

> "Tous les opérandes de type *char* ou *short* sont convertis en *int*, et ceux de type *float* en *double*.
> Puis, si l'un des opérandes est de type *double*, les autres sont convertis en *double* et le résultat sera de ce type.
> Sinon, si l'un des opérandes est de type *long*, les autres sont convertis en *long* et le résultat sera de ce type.
> Sinon encore, si l'un des opérandes est de type *unsigned*, les autres sont convertis en *unsigned* et le résultat sera de ce type.
> Autrement les opérandes doivent être tous de type *int* et le résultat sera de ce type [Reference Manual p. 4-5]".

Les conversions explicites se font à l'aide du cast. Il a la forme suivante : *(nom de type) expression*.
Plus précisément, la valeur renvoyée par *expression* est **interprétée** comme étant de type *(nom de type)*. Dans aucun cas, le type de *expression* n'est changé. L'opérateur de cast *(nom de type)* est un opérateur unaire. Sa priorité est identique aux autre opérateurs unaires et il suit la règle d'associativité à droite.

Quelques exemples d'utilisation sont :

```
(char) (3 - 3.14 * x)
k = (int) ((int) x + (double) i + j)
x = sin((double) n);
```

A l'aide du cast, on est assuré que l'argument de *sin()* est de type double. Remarquez que lors de l'appel la valeur de *n* dans la fonction appelante n'est pas changé, le cast force l'interprétation du type de *n* en un type *double*, cette interprétation est limitée à la fonction *sin()*.

3.3 Les rupture de séquences

Le langage C dispose de diverses instructions de rupture de séquence, trois existent sous forme de mots réservés : *return()*, *break*, *continue* et *goto*. Le "*exit()*" est un appel système.

Une fonction transmet son résultat au programme appelant à l'aide de l'instruction *return(expression);* Si nécessaire, l'expression est convertie, comme pour une affectation, dans le type de la fonction dans laquelle elle se situe.

```
inverse(x)
int x;
{
     return(-x);
}
```

Break permet de sortir prématurément et proprement d'une structure de contrôle.

```
while(1) {
     scanf("%lf", &x);
     if (x < 0.0)
          break;          /* sortir de la boucle si la
                          valeur est négative /*
     printf("%lf\n", x);
     }
```

scanf("%lf", &x); permet de lire un caractère tapé au clavier et *printf("lf\n", x);* permet de l'afficher.

L'instruction *continue* peut seulement être employée dans les boucles *while*, *do* et *for*. Elle provoque un branchement à la fin du corps de la boucle. L'exemple suivant ne traite que les éléments positifs d'un tableau.

```
for (i = 0; i < n; i++) {
     if (a[i] < 0)  /* éliminer les éléments négatifs */
          continue;
          ...  /* traitement des éléments positifs */
     }
```

L'instruction *goto* est employé avec des étiquettes. L'espace de validité des étiquettes est la fonction toute entière dans laquelle elles apparaissent, à l'exclusion de sous-blocs dans lesquels on a redéclaré le même identificateur [KERNIGHAN B. W. & RITCHIE D. M. p. 199].

Exit() est une fonction appartenant à la bibliothèque standard (la directive *#include <stdio.h>* doit faire partie du programme). Elle met fin au programme, ferme les fichiers ouverts et renvoie au système d'exploitation ou programme qui a appelé le programme arrêté, la valeur contenue dans les parenthèses. 0 indique généralement une terminaison normale, tandis que tout autre valeur indique une erreur.

3.4 Les commandes

3.4.1 **La commande conditionnelle**

En C, la condition s'exprime de la manière suivante :

if (expression) instruction-1 [else instruction-2]

où l'exécution de la branche *alors* ou de la branche *sinon* va dépendre de l'évaluation de *expression* : si le résultat est non nul, on exécutera *instruction-1*, sinon on effectuera *instruction-2* (le résultat est alors nul).

De manière tout à fait classique, s'il y a plusieurs instructions dans la partie *alors* ou la partie *sinon*, on construira un *bloc* [TOMBE, Karl p. 10].

Quand il y plusieurs conditions imbriquées et qu'il y a ambiguïté sur un *else*, il est associé au dernier *if* sans *else* qui le précède. Par exemple, dans :

```
if (n > 0)
     if (a > b)
          z = a;
     else
          z = b;
```

le *else* correspond au second *if* ce qui analogue au Pascal.

3.4.2 La commande switch

L'instruction *switch* correspond un peu au CASE en Pascal. est un moyen approprié pour décider dans une instruction à choix multiple. A cette fin, elle teste si une expression prend une valeur parmi un ensemble de constantes et fait le branchement en conséquence. Il faut toutefois savoir qu'il se comporte comme un branchement calculé, et que par conséquent il ne faut surtout pas oublier de mettre les break aux endroits nécessaires :

```
switch (expression) {
     case constante-1 : suite d'instructions break;
     case constante-2 : suite d'instructions break;
     ...
     case constante-n : suite d'instructions break;
     default : suite d'instructions
     }
```

Si on ne met pas de *break*, l'exécution va continuer à la suite au lieu de sortir du *switch*. Il y a parfois des cas où c'est l'effet souhaité.

3.4.3 Les commandes itératives

Le langage C dispose des instructions d'itérations classiques : deux boucles de type *while* et une boucle *for*.

a. La boucle "while"

Voici la première construction répétitive :

```
while (expression)
        instruction
```

C'est la structure TANT-QUE classique. Dans cette construction, on évalue l'*expression*. Si elle est non nulle, on exécute *instruction* puis on regarde à nouveau la valeur d'*expression*. Le cycle continue jusqu'au moment où la valeur d'*expression* devient nulle.

b. La boucle "do ... while"

Une variante de la structure itérative est :

```
do instruction while (expression)
```

Cette forme de boucle permet d'effectuer l'instruction (ou le bloc) une première fois avant le premier test sur la condition d'arrêt.

c. La boucle "for"

L'instruction *for* dont la structure ci-dessous :

```
for(expression1; expression2; expression3)
            instruction;
```

est équivalente à

```
expression1
while(expression2) {
        instruction
    expression3;
};
```

Sur le plan grammatical, les 3 composants d'un *for* sont des expressions. En général *l'expression1* et l'*expression3* sont plutôt des affectations ou des appels de fonctions, et l'*expression2* une expression de relation. On peut ne pas écrire une des trois composants, mais il faut laisser les points-virgules. Si le test *expression2* est absent, on le considère comme toujours positif [KERNIGHAN B. W. & RITCHIE D. M. p. 56].

On a déjà vu l'emploi de *break* dans les structures conditionnelles. On peut l'utiliser également dans une itération pour sortir sans passer par la condition d'arrêt. Par exemple, une boucle qui lit un caractère en entrée (par la fonction *getchar*) et qui s'arrête sur la lecture du caractère *'&'* :

```
for(;;) if((c = getchar()) == '&') break;
```

Dans ce dernier exemple, le test "*expression2*" est absent. Comme, on le considère comme toujours positif, la boucle est donc sans fin et elle doit donc comprendre une instruction *break* ou un *return* pour s'arrêter.

3.5 Les fonctions

Les fonctions permettent de découper un long programme en plusieurs petites sections et de structurer ainsi la solution informatique. En plus, les fonctions offrent l'avantage qu'elles puissent réutiliser les résultats calculés dans d'autres fonctions.

3.5.1 **Le mécanisme de passage de paramètres**

Dans le langage C, toute fonction retourne une valeur. Cette valeur peut être utilisée ou non. La valeur retournée est de type *int* par défaut (une fonction est donc par défault de type *int*), mais on peut typer la fonction. Toutefois un type particulier, *void* (ex : *void f();*), permet d'indiquer qu'une fonction ne retourne pas de valeur, ou plutôt que la valeur retournée ne doit pas être prise en compte.

L'unique type de passage des paramètres est le passage par valeur. Cela implique que la fonction appelée reçoit les valeurs de ses arguments par l'intermédiaire de variables provisoires et non pas leurs adresses [KERNIGHAN B. W. & RITCHIE D. M. p. 26]. L'appel "par adresse" comme **mécanisme de passage de paramètres** n'existe pas en C.

Si l'on veut qu'une fonction modifie la valeur d'une variable à l'extérieur de cette fonction, il faut lui passer l'adresse de cette variable.

Dans l'exemple 2.1, la fonction *main()* appelle la fonction *swap()* et passe deux paramètres *i* et *j*. Ils sont précédés par l'opérateur *&* qui renvoie l'adresse de l'opérande. Dans l'appel de la fonction, *x* et *y* reçoient les adresses de *i* et de *j* respectivement. A l'intérieur de la fonction, ils sont déclarés comme des pointeurs. Leur initialisation provoque que chacun pointe sur *i* et *j* respectivement.

```
main()
{
int i, j;

i = 5;
j = 9;
swap(&i, &j);
}
```

```
swap(x, y)
int *x, *y;
{
int z;

z = *x;
*x = *y;
*y = z;
}
```

Exemple 2.1 : Illustration de passage d'une adresse lors de l'appel d'une fonction

Cependant, il existe une exception au passage des arguments par valeur. Quand le nom d'un tableau constitue l'argument d'une fonction, la position du premier élément est transmise. Les éléments ne sont pas recopiés. La conséquence est qu'en fait les tableaux sont transmis par

variable (cfr en Pascal, la déclaration d'un paramètre "var" indique que ce paramètre recevra une adresse) [KERNIGHAN B. W. & RITCHIE D. M. p. 69].

3.5.2 **Des pointeurs sur des fonctions**

Les pointeurs ne sont pas uniquement employés avec les structures de données, ils peuvent aussi bien pointer sur des fonctions, comme il est illustré dans l'exemple 2.2.

1° *int (*f)();* — *f* est un pointeur vers une fonction qui renvoie un entier.

L'appel de la fonction dans l'instruction *y = (*f)(x);* affecte à px un entier, calculée dans la fonction *(*f)()*. *f* doit naturellement pointer sur une fonction.

La déclaration *int (*f)();* est différente de la déclaration *int *f();* .

2° *int *f();* — *f* est une fonction qui renvoie un pointeur vers un entier.

Exemple 2.2 : Deux combinaisons de pointeurs avec des fonctions

Les pointeurs vers les fonctions peuvent être manipulés comme des pointeurs ordinaires, ils peuvent être placés dans un tableau, ils peuvent être transmis comme paramètres à d'autres fonctions.

L'exemple 2.3 fait partie d'une fonction de tri.

```
tri(tab, n, comp)
int *tab;
int n;
int (*comp)();
{
     int i,j;
     i = j = 0;
     while (j <= n) {
          ...
          if ((*comp)(tab[i], tab[j])) <= 0) {
                    ... }
               else {
                    ... }
          j++;
          }
}
```

Exemple 2.3 : un appel de fonction par l'intermédiaire d'un pointeur

La fonction *tri(tab, n, comp)* reçoit trois paramètres. Le premier paramètre *tab* est interprété comme l'adresse d'un entier. Le deuxième paramètre contient le nombre d'éléments à trier. Ces éléments sont rangés en mémoire à partir de l'adresse *tab*. Le troisième paramètre *comp* est interprété comme l'adresse d'une fonction.

L'appel de la fonction de comparaison *comp* se fait avec *(*comp)(tab[i], tab[j])*. La construction *(*comp)* permet d'accéder à la fonction. L'interprétation de *tab[i]* et de *tab[j]* fournit les adresses des éléments *i* et *j* du tableau à la fonction de comparaison.

Afin de permettre un appel correct de la fonction *tri(tab, n, comp)*, il faut que *comp()* soit déclarée dans la fonction appelante de la façon : *int comp();*. Au moment de l'appel de *tri(tab, n, comp);*, le compilateur fait en sorte que l'adresse soit transmise dans le paramètre *comp*.

Remarquez que le compilateur n'a aucune information sur le nombre et le type des arguments de la fonction *int comp()* lors de sa déclaration dans la fonction appelante [KERNIGHAN B. W. & RITCHIE D. M. p. 112].

3.5.3 **Un exemple de déclarations complexes**

char *(*(*var)())[10]; est un autre exemple de déclarations complexes combinant des pointeurs, des tableaux et des fonctions :

*char *(*(*var)())[10];* — *var* est un identificateur de type pointeur vers une fonction qui renvoit un pointeur vers un tableau de 10 éléments qui sont des pointeurs vers des valeurs de type caractère

Le schéma suivant peut faciliter la compréhension :

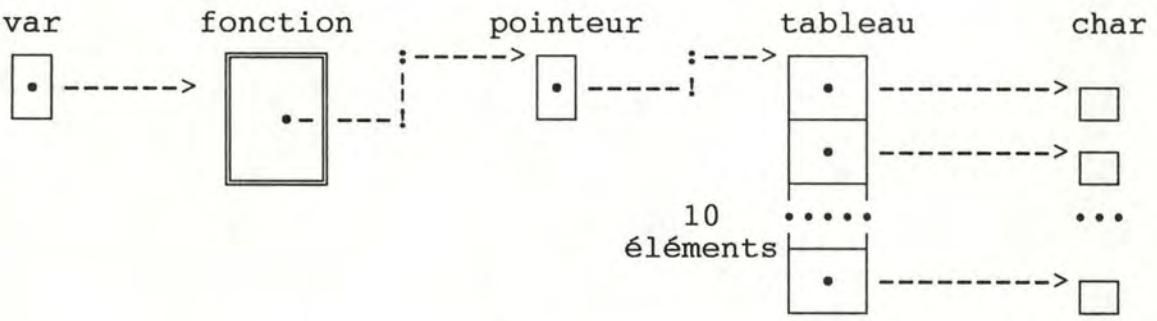


Figure 2.5 : une représentation de la structure virtuelle correspondant à l'exemple *char *(*(*var)())[10];*

La lecture de la fonction de l'intérieur vers l'extérieur facilite la compréhension.
*(*var)* symbolise la fonction et on peut remplacer *(*var)* par *f*.
Dans **(*f())[10]*, *(*f())* renvoie un pointeur vers un élément d'un tableau de 10 pointeurs (**(*f())[10]* est identique à **((*f())[10])* car les *[]* sont plus prioritaires que * donc les *[]* sont d'abord interprétés).
On peut remplacer *(*f())[10])* par une adresse (qui est contenue dans le tableau), par exemple *i*, alors *char *i* représente un pointeur vers un caractère.
L'ensemble de cette construction est de type caractère.

4 Le préprocesseur

Tout compilateur C est muni d'un préprocesseur qui modifie le code source de départ en tenant compte des directives qui lui sont données. Le précompilateur est un outil de "traitement" de texte qui considère uniquement les aspects purement formel d'un programme. Il n'associe aucune sémantique aux constructions syntaxiques. Le préprocesseur est invoqué automatiquement avant que le compilateur commence le travail d'analyse syntaxique et traduction en code machine.

Les commandes destinées au préprocesseur doivent être précédées d'un dièse (#).

La directive

#include <nom de fichier>

permet d'inclure au code source le contenu du fichier donné en paramètre. Ce peut être un autre fichier contenant des fonctions en C, mais comme la compilation séparée est plus intéressante pour cela, ce sera la plupart du temps des fichiers de définitions de variable externes.

Le préprocesseur permet à de la directive *#define* de définir des macro-commandes. On peut définir au début d'un programme un nom symbolique ou une constante symbolique et lui affecter une manière automatique dans tout le programme.

Par exemple :

```
#define     MAX         1000
#define     max(x,y)    ((x < y) ? (y) : (x))
```

Dans le programme *MAX* sera remplacé par *1000* et *max(x,y)* par *((x < y) ? (y) : (x))*. Les noms symboliques traités par le précompilateur sont généralement écrits en majuscules pour les distinguer des noms de variables qui sont écrit en minuscules et traités par le compilateur lui-même.

D'autres instructions sont :

#undef
#ifdef
#ifndef

Elles permettent respectivement de défaire une définition, de tester si une définition a été établie ou n'a pas été établie et d'agir selon le résultat du test.

5 Les Entrées-Sorties en C

Les entrées/sorties ne font pas partie du langage C. Toutefois, il existe une bibliothèque standard.

Cette bibliothèque définit un ensemble de fonctions qui fournit un système d'entrées/sorties standard pour les programmes écrit en langage C. Le fonctions sont destinées à former un interface commode de programmation et n'effectuent que les opérations traitées par la plupart des systèmes d'exploitation. La description suivante ne veut guère être complète, pour une discussion détaillée, le lecteur peut se référer au manuel UNIX.

Quand un programme commence, le système d'exploitation ouvre automatiquement trois fichiers et, pour chacun, il fournit un pointeur de fichier. Ces fichiers correspondent à l'entrée standard, la sortie standard et la sortie d'erreur standard, les pointeurs correspondants sont appelés *stdin*, *stdout*, *stderr*.

Tout fichier source qui fait référence à une fonction de la bibliothèque standard, doit contenir au début la ligne :

#include <stdio.h>

Le fichier *stdio.h* définit certaines macro-instructions et certaines variables utilisées par la bibliothèque des entrées/sorties.

La fonction *getchar()* lit un caractère à la fois provenant de l'entrée standard, généralement un terminal. La fonction *putchar(c)* (c est un entier) transmet le caractère *c* à la sortie standard.

Les fonctions *printf()* et *scanf()* permettent de faire des entrées/sorties formatées. *Printf()* permet d'écrire sur la sortie standard. Elle dispose de la syntaxe suivante : *printf(control, arg1, arg2, ...)*.

Enfin, l'environnement du langage C dispose de fichiers. L'accès est à nouveau effectué avec des pointeurs. Les fonctions *open()*, *fopen()*, *close()* et *fclose()* permettent respectivement d'ouvrir et de fermer des fichiers. *getc() et putc()* respectivement lisent et écrivent un caractère à la fois. L'accès aléatoire se fait via *seek()* et *lseek()*.

6 La compilation conditionnelle et séparée

Le langage C se prête particulièrement bien à la compilation séparée. Il est fortement conseillé de travailler avec de petits fichiers, et même de se faire des bibliothèques de fonctions. Les règles habituelles de la compilation séparée doivent bien sûr être respectées, en particulier, toute variable externe doit être définie dans un fichier et déclarée avec le mot-clé *extern* dans les autres fichiers où on l'utilise.

Chapitre III :

La sémantique du langage C

1 Introduction

Les principes sémantiques que les langages conventionnels doivent satisfaire, seront décrit dans ce chapitre. La sémantique présentée est informelle et est basée sur des principes dénotationnels.

Certains aspects de la sémantique du langage C ont déjà été décrits de façon intuitive dans le deuxième chapitre. Dans ce chapitre, un nombre réduit de concepts sémantiques seront expliqués qui permettront de préciser l'effet sémantique de l'exécution de chaque instruction.

D'abord, on étudiera les concepts qui permettent de modéliser la mémoire dans laquelle les données se trouvent. La notion de "contexte du calcul" structurée "environnement", "état de la mémoire" et "continuation d'une instruction" sera expliquée. Ensuite, on étudiera les instructions C qui se trouvent dans les cinq classes syntaxiques déjà mentionnées, en employant la notion de contexte de calcul d'une instruction.

Les principes du modèle sémantique présenté, sont basés sur les concepts de "state of the computer store", "environment" et "continuations" comme ils sont expliqués dans "*Principles of Programming Languages*" de TENNENT R. D. et "*Denotationnal semantiques*" de STOY. D'autres articles de référence sont "*An introduction to formal semantics*" de STRACHEY, "*The next 700 programming Languages*" de LANDIN et "*The semantics of Scheme*" de William CLINGER.

2 La syntaxe concrète et abstraite

Un programme est un texte composé d'un ensemble de phrases. Ces phrases sont construites avec des mots réservés ou des mots définis par le programmeur. Elles sont accompagnées de diverses constructions syntaxiques. En autres termes, programme est exprimé en une syntaxe concrète. Les petits programmes en C dans le deuxième chapitre forment des exemples de cette syntaxe concrète.

Dans ces constructions syntaxiques, on peut regrouper les instructions qui ont des propriétés similaires, par exemple grouper toutes les déclarations, toutes les instructions de contrôle, etc ... . En travaillant ainsi, on construit la syntaxe abstraite d'un langage. La syntaxe abstraite indique les catégories syntaxiques qui sont disponibles dans un langage, mais elle ne se préoccupe pas des aspects qui indiquent si les chaînes de caractères sont "bien formée" ou pas [LANDIN, P. p. 159].

On a identifié cinq classes syntaxiques dans les langages conventionnels (les déclarations et définitions, les expressions, les procédures, les commandes et les ruptures de séquence). Le modèle sémantique qu'on construira dans les sections suivantes, sera employé pour exprimer la signification de ces classes syntaxiques.

Le modèle sémantique considéré, est basé sur la sémantique dénotationnelle. Cette approche associe des valeurs abstraites aux constructions syntaxiques. Cette association est modélisée par des fonctions mathématiques, qui sont généralement définies de façon récursives. La valeur d'une construction est spécifiée en termes de valeurs dénotées par les composants syntaxiques [STOY J. p. 13].

3 L'établissement du modèle sémantique

La mémoire constitue un composant essentiel des ordinateurs du type Von Neumann. Supposons qu'on dispose de la séquence suivante d'instructions impératives,

I_1; I_2

Pour que I_2 puisse employer les résultats que I_1 a stocké en mémoire, il faut que I_2 soit mis au courant de cet endroit. Comment peut-on correctement modéliser "cet endroit" et comment peut-on I_2 accéder aux informations stockées en mémoire ?
Plus généralement, I_2 doit recevoir un "environnement" et un "état de la mémoire" pour pouvoir s'exécuter correctement.

L'exemple I_1; I_2 est basé sur une hypothèse implicite, c'est-à-dire on suppose que l'exécution de ces instructions se fait de façon séquentielle. Mais est-ce que cela est toujours le cas ? Il est existe des mécanismes d'itération et de sélection. Comment peut-on traiter la sémantique de ces instructions, sachant en plus que le C dispose de plusieurs constructions de rupture de séquence ?
Le principe de la "continuation" d'une instruction permettra d'interpréter correctement le sens de ces constructions syntaxiques.

Les notions d'environnement, d'état de la mémoire et de continuation sont les piliers de base du modèle sémantique qui sera développé pour déterminer le sens des instructions C.

3.1 L'environnement et l'état de la mémoire

Dans la théorie mathématiques, on emploie des variables pour dénoter des objets. Ces objets peuvent être des nombres, des caractères, etc ... . Si l'on suppose qu'à un moment donné une variable dénote un seul objet, on peut considérer cette dénotation comme une relation entre la variable et l'objet.

En informatique, la mémoire est composée de plusieurs cellules. Ces cellules peuvent contenir des chaînes de bits. A un moment donné, une cellule ou un emplacement peut contenir une seule chaîne de bits. Ici à nouveau, on peut

considérer ce fait de "contenir quelque chose" comme une relation entre une cellule particulière et la chaîne de bits qu'elle contient.

On peut facilement constater qu'il existe des similarités entre la relation variable - objet d'un côté et la relation cellule - chaîne de bits d'un autre côté. Les deux relations contiennent une notion de "variabilité temporelle" c'est-à-dire que l'objet dénoté par la variable et la chaîne contenue par la cellule contient sont fixes à un moment donné mais peut varier dans le temps. Mais il y a une différence de niveau d'abstraction, les variables et les objets sont des concepts abstraits, les cellules et les bits sont des concepts concrets.

Si l'on approfondit les correspondances entre les variables et les objets d'un côté et les cellules et les chaînes de bits de l'autre, on découvre des liens supplémentaires. Il existe notamment deux relations qui traversent la séparation entre le niveau abstrait et concret.

La première relation permet d'établir très facilement un lien entre l'objet et la chaîne de bits : l'un est une représentation de l'autre. Par exemple, le code ASCII ou EBCDIC est une représentation en binaire de 256 caractères qui sont couramment employés.

La deuxième relation établit un lien entre les variables et les emplacements en mémoire. Une variable dénote un seul objet à un moment donné. Un emplacement contient une seule chaîne de bits à un moment donné. Une variable en mathématiques est identique à un emplacement en mémoire au différence de niveau prêt.

On peut employer la représentation suivante pour rendre ces concepts plus concrets.

Figure 3.1 : la relation entre environnement et l'état de la mémoire

Les relations qui viennent d'être schématisées, doivent être définies plus précisément.

L'ensemble des relations entre les variables et leurs emplacements en mémoire fait partie de l'**environnement**. Les relations entre les emplacements ou les adresses en mémoire et les chaînes de bits qu'elle contiennent font partie de l'**état de la mémoire**.

En informatique, on parle plutôt d'identificateurs que de variables. Un **identificateur** est le nom d'une cellule mémoire dont la valeur peut être définie ou redéfinie par le programmeur [TENNENT, R. D. p. 11].

Le concepts d'environnement et d'état de la mémoire sont basés sur la sémantique dénotationnelle. Dans cette sémantique, ces notions sont employées comme paramètres de la fonction sémantique. Cette fonction donne la signification d'une construction syntaxique [STOY J. p. 253].

La sémantique dénotationnelle associe à l'état de la mémoire encore d'autres fonctions comme par exemple une fonction qui pour une adresse donnée, indique si cet emplacement est employé ou non, une fonction qui renvoie le contenu d'une cellule, etc ... [STOY J. p. 285-286]. Ces constructions mathématiques sont uniquement utiles sur le plan théorique et ne seront pas considérées en profondeur.

L'environnement contient tous les renseignements qui peuvent être déduits d'un programme au moment de la compilation. Il s'agit surtout de tables d'adresses des variables. L'état de la mémoire est seulement connu à l'exécution du programme. Concrètement, l'état de la mémoire est représenté par les diverses structures de données construites dans un programme [STOY J. p. 231].

3.2 La continuation d'une instruction

Les langages conventionnels en général et le C en particulier disposent de différents mécanismes pour déterminer l'ordre dans lequel les instructions impératives doivent être exécutées. Comment peut-on déterminer la sémantique de ces différents ordres d'exécution ?

On distingue dans les constructions impératives des commandes et des ruptures de séquence.

La séquence d'exécution des commandes est basée sur la loi de la composition fonctionnelle, où $(h \circ g)(x) \equiv h(g(x))$. Supposons que C_1 et C_2 sont deux commandes, alors la construction syntaxique "C_1; C_2" signifie $C_2 \circ C_1$, c'est-à-dire C_2 s'exécute après C_1.

La loi de composition fonctionnelle est trop stricte et n'est pas suffisamment flexible pour déterminer la sémantique de toutes les constructions impératives. Elle ne permet pas d'exprimer la signification des ruptures de séquence. Ceci est illustré par l'exemple suivant :

{l:C₁; ...; goto l; ...}; C₂;

Dans cet exemple, le goto provoquera un rupture de séquence en dehors de la construction et C_2 ne sera jamais exécutée [TENNENT, R. D. p. 24].

On pourrait répondre, les "goto", il faut éviter de les employer. On n'a pas besoin d'une explication de sa signification.

Mais comment faut-il alors exprimer la sémantique de la construction suivante :

```
abs(x)
int x;
{
     if (x >= 0) return(x);
     x = -x;
     return(x);
}
```

Si, dans un appel de cette fonction, la valeur du paramètre actuel *x* est effectivement supérieur ou égal à zéro, l'instruction *x = -x;* ne sera jamais exécutée puisque la fonction aura déjà retourné son résultat. L'exemple est un peu forcé dans ce sens qu'on pourrait transformer le contenu du bloc en l'instruction suivante "*if (x >= 0) return(x) else return(-x);*". La sémantique de cette construction peut être exprimée grâce à la composition fonctionnelle (il ne reste qu'une seule instruction dans le bloc, on peut considérer cela comme une composition avec une instruction vide).

Mais cette approche signifierait que pour chaque rupture de séquence, qu'il faudrait appliquer une ou plusieurs transformations syntaxiques pour aboutir à une construction dont on peut exprimer la sémantique, à l'aide de la loi de la composition fonctionnelle. Cette démarche impliquerait que cette transformation syntaxique existe effectivement. En plus, elle est contre l'approche dénotationnelle qui essaie d'exprimer la sémantique d'une construction en termes de ses composants comme ils sont écrits sans devoir appliquer des transformations [STOY J. p. 252].

On a besoin de la notion de **continuation** pour pouvoir correctement exprimer la sémantique du séquencement des instructions dans un langage. Intuitivement, on peut comprendre la **continuation** d'une instruction comme une indication de ce que le programme doit faire avec les résultats intermédiaires pour obtenir le résultat final (du programme) [TENNENT, R. D. p. 224].

La continuation d'une instruction peut être considérée comme **l'effet dynamique** de ce qui reste à exécuter d'un programme après la terminaison normale de cette instruction. Il ne s'agit pas nécessairement de l'instruction suivante dans le texte d'un programme.

Plus rigoureusement, la continuation d'une instruction sera définie comme une fonction qui pour chaque instruction (appartenant à n'importe quelle classe syntaxique) donne la suite de cette instruction **en termes de l'instruction suivante à exécuter**, la fonction de continuation passe à l'instruction à exécuter les changements apportés dans le contexte par l'instruction précédente.

Concrètement, la continuation correspond au "control stack" dans les langages comme le Pascal ou le C [Clinger W, p. 226].

3.3 Le contexte d'une instruction

Dans la section précédente, les notions d'**environnement**, d'**état de la mémoire** et de **continuation** ont été présentées. Dans leur ensemble, ils définissent **le contexte d'une instruction.**

Par la suite, la sémantique d'une instruction sera toujours déterminée par rapport à son contexte. La signification des instructions appartenant aux cinq classes syntaxiques sera spécifiée en fonction des notions d'environnement, d'état de la mémoire et de continuation.

Le concept de "contexte d'une instruction" est une traduction de ce que Tennent appelle "the state of the computation". Dans l'optique de cet auteur, "the state" a deux composants : "the environment" et "the store" [TENNENT, R. D. p. 16]. Stoy définit les mêmes concepts, avec d'autres termes. Il emploie aussi "the environment", mais au lieu de parler de "store", il utilise "the state of the computer store" [STOY J. p. 284]. Le terme "the environment" a été traduit en "l'environnement", "the state of the computer store" ou "the store" en "l'état de la mémoire" et "the continuation" en "la continuation".

En même temps, on peut expliciter l'emploi du mot "instruction" dans ce chapitre. Il est employé comme une sorte de terme générique remplaçant l'énumération des cinq classes syntaxiques.

4 Le contexte d'une instruction et les classes syntaxiques

Les cinq classes syntaxiques ont déjà été mentionnées régulièrement dans ce mémoire, sans qu'on les ait décrites rigoureusement. Dans cette section, on définira la sémantique d'une instruction en fonction des effets de

l'exécution de cette instruction par rapport à son contexte. Plus loin dans le mémoire, ces définitions seront affinées et nuancées pour correctement exprimer le comportement de l'exécution d'une instruction en C par rapport à son contexte.

Par la suite, on emploiera les conventions de notation suivantes :
- D désignera les déclarations et les définitions,
- E " les expressions,
- C " les commandes,
- P " les procédures,
- J " les ruptures de séquence,
- { } indiquent respectivement le début et la fin d'un bloc.

Ces notations sont basées sur celles que Tennent emploie dans "Principes of Programming Languages" [TENNENT, R. D. p. 13, 14, 15].

Dans les divers schémas qui se trouvent dans ce chapitre, "env" signifiera environnement, "état" signifiera état de la mémoire et la continuation sera représentée comme la fonction "cont()".

4.1 Les déclarations, les définitions et les blocs

4.1.1 Le contexte des déclarations et des définitions

a. Les déclarations et les définitions

Une définition crée un nom symbolique pour un objet ou une valeur. Après son exécution, on peut employer le nom qu'on vient de définir, au lieu de mentionner l'objet ou la valeur. Par exemple, un programmeur pourra définir "pi" comme nom symbolique pour la valeur "3.14". Ensuite, dans une assignation, il peut écrire "x = r * r * pi;" au lieu de "x = r * r * 3.14;".

Il existe une différence fondamentale entre une définition et une déclaration. Ces deux mécanismes établissent un lien entre un nom et un objet.

Une déclaration est caractérisée par le processus suivant : le compilateur réserve une cellule mémoire et lui associe le nom défini par le programmeur. Le contenu de cette cellule peut être modifié grâce à une commande. Une déclaration implique la notion de variabilité temporelle (cfr la section 3.1 l'environnement et l'état de la mémoire).

Par contre, dans les définitions, cette variabilité temporelle n'existe pas. Il s'agit d'une relation d'identité entre le nom et l'objet. L'état de la mémoire n'intervient pas, car la chaîne de bits n'est pas contenue dans une cellule mémoire, qui est accessible et modifiable par le programmeur. Le modèle sémantique "contexte d'une instruction" n'est donc pas applicable. Comment peut-on résoudre cette difficulté sémantique ?

Il faut considérer une définition comme ayant uniquement un effet sur l'environnement. Un lien entre le nom et la valeur est créé sans passer par un emplacement en mémoire. La relation "état de la mémoire" n'intervient pas donc pas dans le cas d'une définition. En pratique, après avoir rencontré une définition, le compilateur remplace toutes les occurrences du nom symbolique par sa valeur. Il s'agit donc uniquement d'un remplacement au niveau texte du programme.

Le schéma 3.1 a été repris et modifié en tenant compte avec les nouvelles relations. Le terme "variable" a été remplacé par "nom" pour couvrir le cas d'une définition.

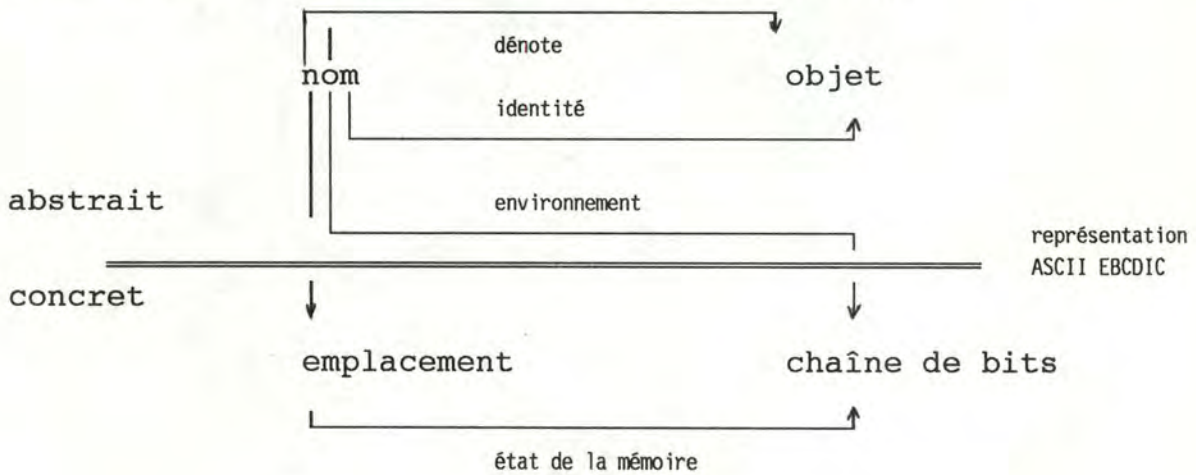


Figure 3.2 : la relation entre environnement et l'état de la mémoire dans le cas d'une déclaration et définition

4.1.2 La structure de bloc

Le Pascal, l'Algol, l'ADA et le C sont des langages qui implémentent des structures de bloc. Un bloc est un mécanisme qui permet de regrouper plusieurs instructions en une seule action [GHEZZI & JAZAYERI p. 28].

Dans les langages de programmation, les déclarations et les définitions faites au début d'un bloc, ont une portée limitée à ce bloc. Il faut distinguer la portée temporelle de la portée régionale. La portée temporelle indique le temps pendant lequel une variable existe et est manipulable par le programme. La portée régionale comprend le texte du programme pendant lequel la variable existe.

Dans le langage C, un bloc a la forme suivante :

```
{
D
C
}
```

Pour le moment, on restera assez vague sur le contenu exact de D. C est une abréviation pour les commandes.

L'imbrication des blocs peut diviser la portée régionale d'une variable. La redéclaration d'une variable dans la couche inférieure d'un bloc divise la portée régionale. Cela est illustré par l'exemple 4.1. La portée régionale de la variable *x* est indiquée par le trait "|".

```
}
int x;                  |
x = 4;                  |
     }                  |
     int x;             |
     x = 5;             |
     }                  |
printf("%d\n", x);      |
}
```

Exemple 3.1 : une illustration de la portée régionale d'une variable

Dans le langage C, les définitions sont traitées par le précompilateur. Il considère le texte d'un programme uniquement comme un concept formel, purement syntaxique (sans signification) et linéaire sans mécanisme de structuration. Concrètement, après la séquence "*#define MAX 100*" toutes les occurrences de *MAX* seront remplacées par *100* jusqu'à la fin du fichier ou jusqu'au moment de la séquence "*#undef MAX 100*".

La notion de portée temporelle et régionale n'existe donc pas pour les définitions. En termes générales, la structure de bloc existe uniquement pour les déclarations en C et pas pour les définitions.

Les déclarations et définitions sont exécutées par rapport à un contexte complet. En termes sémantiques, le précompilateur traite uniquement le texte du programme et dans ce sens les définitions en C modifient uniquement environnement. Les déclarations préparent l'environnement et l'état de la mémoire pour être manipulé par le programme. La continuation d'une déclaration est une commande. L'effet de l'exécution d'une déclaration et d'une définition est représenté dans la figure 3.3.

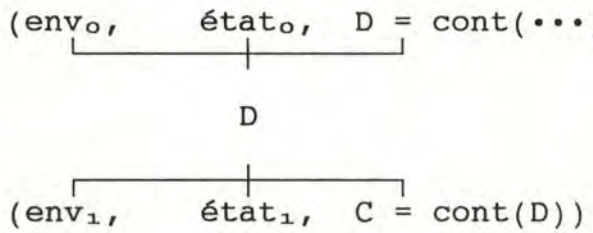


Figure 3.3 : l'effet de l'exécution d'une déclaration

4.2 Les commandes

4.2.1 **Le contexte des commandes**

Dans la section précédente, la sémantique des déclarations et des définitions a été établie. Les déclarations et les définitions construisent l'environnement et préparent l'état de la mémoire pour être consulté et modifié par d'autres types d'instructions comme les expressions, les instructions de branchement, les fonctions et les commandes. Dans cette section, un de ces mécanismes sera étudié en détail, il s'agit de la catégorie syntaxique des commandes.

La définition d'une commande est inspirée sur la sémantique dénotationnelle. Dans cette approche de la sémantique, on définit une commande comme une fonction de transformation de l'état de la mémoire. Cette fonction reçoit un contexte complet comme argument, et produit un nouvel état. La continuation d'une commande est à nouveau une commande qui reçoit l'état de la mémoire qui est modifié par la commande précédente [TENNENT, R. D. p. 12 et STOY J. p. 227]. L'environnement n'est pas modifié par une commande.

Concrètement, changer l'état de la mémoire implique à modifier une ou plusieurs valeurs contenues dans des cellules en mémoire. On peut schématiser l'effet de l'exécution d'une commande de la façon suivante :

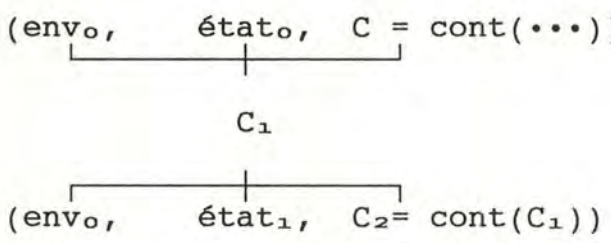


Figure 3.4 : l'effet de l'exécution d'une commande

Dans les langages conventionnels, on distingue les commandes suivantes : l'assignation, la séquence, le groupement de commandes (par des begin et des end ou { et }), la sélection

et la répetition. Les quatre derniers catégories de commandes sont souvent regroupées sous le nom de structures de contrôle. Elles permettent entre autres de combiner les effets de l'assignation [TENNENT, R. D. p. 79].

Syntaxiquement, il existe deux types de commandes : des commandes simples et indécomposables (par exemple : l'instruction nulle comme le "skip" en PASCAL) et des commandes composées, qui sont souvent constituées d'une expression et d'une ou plusieurs commandes.

L'effet sémantique de l'exécution des différentes commandes sera expliqué dans les sections suivantes. Plusieurs types de commandes ont des expressions comme composants. Pour le moment, on supposera que les expressions renvoient une valeur et un état de la mémoire. Les raisonnements sémantiques derrière ces hypothèses seront expliqués dans la partie sur les expressions.

4.2.2 La commande nulle

La commande nulle de forme ";". peut être utilisée comme corps d'une boucle [REFERENCE MANUEL p. 19].

4.2.3 La commande de l'assignation

La commande d'assignation fait évaluer l'expression qui se trouve à droite et l'affecte à la partie gauche. Une discussion détaillée se trouve dans la section 4.4.

4.2.4 La séquence et le groupement de commandes

La composition séquentielle est la structure de contrôle la plus simple. En C, elle est exprimée par la séquence textuelle. La séquence d'instructions "C_1 C_2" indique que "C_2" doit être exécutée après "C_1". En termes sémantiques, la continuation de "C_1" est "C_2", qui reçoit l'état de mémoire modifié par "C_1". La continuation de "C_2" reçoit la continuation de l'ensemble "C_1 C_2". L'environnement n'a pas changé.

L'exécution des deux commandes peut être représentée par le schéma suivant :

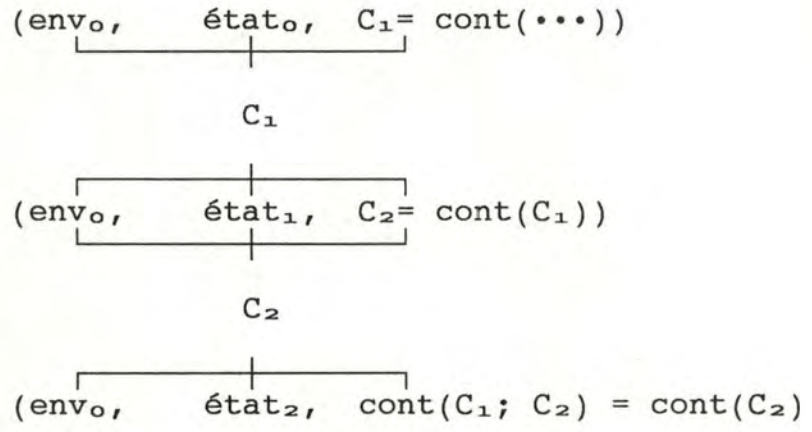


Figure 3.5 : l'effet de l'exécution d'une séquence de commandes

Les accolades { et } sont employés pour regrouper des déclarations et des commandes dans une construction syntaxique qui porte le nom d'une "instruction composée" ou un "bloc". Sur le plan de la syntaxe, un bloc devient équivalent à une seule commande [KERNIGHAN B. W. & RITCHIE D. M. p. 51]. Chaque fois qu'une commande fait partie d'une construction plus élaborée, on peut la remplacer par un bloc comme par exemple, dans *if (E) C_1 else C_2*, chaque occurrence de C peut être remplacée par un bloc,

```
if (E) {
        D
        C
}
else {
        D
        C
}
```

Le modèle sémantique s'applique sans modification sur chacune des constructions à l'intérieur d'un bloc. La continuation de la dernière commande d'un bloc reçoit la continuation du bloc complet. L'état de la mémoire après l'exécution du bloc est égal à l'état après l'exécution de la dernière commande de ce bloc.

4.2.5 La sélection

La construction de sélection permet au programmeur de spécifier qu'un choix doit être effectué entre plusieurs alternatives, et cela en fonction de certains critères [GHEZZI & JAZAYERI p. 174].

Le langage C dispos de deux types de commandes de sélection : le "if" et le "switch".
La commande "*if (E) C_1 else C_2*" peut être décomposée en trois parties : une expression et deux commandes.
L'exécution de la commande conditionnelle dans son ensemble commence avec l'évaluation de l'expression *E* qui renvoie la valeur "vrai" ou "faux" et un état de la mémoire. Le mécanisme de sélection choisira C_1 ou C_2 en fonction de la valeur renvoyée (C_1 dans le cas de la valeur "vrai" ou C_2 dans le cas "faux"). La continuation de l'expression *E* sera donc C_1 ou C_2.

Par conséquent, une des deux commandes est exécutée et par définition, elle transforme l'état reçu de l'évaluation de l'expression *E*. La continuation de C_1 ou C_2 reçoit la continuation de la construction conditionnelle dans son ensemble. Cette continuation est à nouveau une commande [TENNENT, R. D. p. 148].

Plusieurs constructions conditionnelles peuvent être imbriquées en suivant les règles indiquées dans la section concernant le langage C.

L'effet sémantique de la commande "*switch*" peut être expliqué à l'aide de cadre sémantique de la commande "*if*".
L'évaluation de l'expression *E* calcule une valeur qui est passée au mécanisme de sélection, et choisira une commande qui sera la continuation de *E*. Cette commande reçoit l'état

de la mémoire de *E*. Sa continuation est égale à la continuation du "*switch*" dans son ensemble, même si plusieurs commandes de la construction sont exécutées (à cause de l'absence d'un *break*, par exemple). Dans ce cas, on peut considérer cet ensemble de commandes comme une seule.

4.2.6 L'itération

La construction itérative permet de répéter un certain nombre de fois une ou plusieurs commandes. Dans les langages de programmation, on peut souvent distinguer deux types de boucles, il existe des boucles dont le nombre d'itérations est fixe et est déterminé à la compilation (par exemple : le "do" en Fortran et le "for" en Pascal). Dans le deuxième type de boucles, le nombre d'itérations est dynamique et dépend de l'exécution. Dans la plupart des langages, il est connu sous le nom de "while".

En C, il existe trois types de boucles :

La boucle "*while*",	la boucle "*do while*",	la boucle "*for*"
while (E) C	*do C while (E)*	*for(E_1;E_2;E_3) C*

Une généralisation de ces commandes peut être représentée sous la forme suivante :

```
loop
     C1
  while (E)
     C2
repeat
```

[KNUTH, D. E.].

La boucle "*while (E) C*" peut être obtenue de cette construction générale en considérant C_1 comme la commande nulle (*;*) et "*do C while (E)*" en considérant C_2 comme *;*. Malgré son nom, le nombre d'itérations du "*for(E_1;E_2;E_3)*" n'est pas toujours connu à la compilation car une des trois expressions peut être absente.

Ceci implique cette construction peut aussi être appliquée à ce schéma. L'instruction "*for()*" correspond à la boucle générale où C_1 est la commande nulle.

La sémantique du "loop C_1 while (E) C_2 repeat" est décrite de la même façon que la sémantique du if. La commande peut être décomposée en trois parties : une commande, ensuite une expression et pour finir à nouveau une commande. L'exécution du "loop" commence avec C_1 dont sa continuation est E. Elle est évaluée en se basant sur l'état de la mémoire modifié par C_1. L'évaluation de E renvoie la valeur "vrai" ou "faux" et un état de la mémoire. Le mécanisme d'itération sélectionnera en fonction de la valeur renvoyée, C_2 ou la continuation du "loop" dans son ensemble comme continuation de E (dans ce cas, on sort de la boucle). L'état de la mémoire modifié par E, est passé à sa continuation. Si C_2 est exécutée, sa continuation est C_1, qui reçoit l'état de la mémoire modifié par C_2 [TENNENT, R. D. p. 148].

4.3 Les expressions

4.3.1 **Le contexte des expressions**

Dans la section précédente, la sémantique des commandes a été établie. Les commandes modifient l'état de la mémoire. Elles changent donc les valeurs contenues dans les emplacements en mémoire. Mais on ne sait pas avec quels mécanismes qu'on peut produire des nouvelles valeurs pour modifier le contenu des cellules. Dans la discussion sur les commandes, l'hypothèse a été posée qu'une expression produit une valeur et un état de la mémoire, sans qu'on connaisse exactement les raisons pourquoi. Le mécanisme de production d'une valeur est fournie par l'expression. Ceci sera expliqué dans cette section, en même temps l'hypothèse sera justifiée.

Dans la partie sur le langage C, les expressions ont été présentées. Une expression est un procédé arithmétique qui à partir d'un ensemble de valeurs, permet de calculer une seule

valeur. L'importance des règles d'association et de la priorité des opérateurs a été indiquée dans le deuxième chapitre.

Pendant le processus de l'évaluation d'une expression, l'emploi de valeurs intermédiaires est souvent nécessaire. Ces valeurs sont obligatoirement contenues dans des cellules ou des registres. Elles nécessitent donc des modifications dans l'environnement et dans l'état de la mémoire. Il s'agit par exemple de la mise en oeuvre de variables locales à une fonction ou des variables intermédiaires générées par le compilateur. Mais ces modifications dans l'environnement et dans l'état de la mémoire n'ont pas un caractère permanent, elles cessent d'exister après l'évaluation de l'expression.

Ceci n'est pas toujours vrai. Il se peut que l'évaluation ne produit pas uniquement une valeur, mais qu'elle provoque aussi un changement permanent dans l'état de la mémoire après la terminaison de l'évaluation. Plus précisément, le contenu des cellules avant et après l'évaluation d'une expression ne sont pas identiques. Dans ce cas, on dit que l'évaluation d'une expression provoque **un effet de bord** ou **un side effet.**

En tenant compte des caractéristiques précédentes, on peut correctement définir la sémantique d'une expression. A l'exécution, une expression est évaluée par rapport à un contexte complet. On peut considérer une expression comme une fonction qui produit une valeur et un nouvel état. Cette valeur et cet état qui reflète éventuellement les effets de bord, sont passés à la continuation de l'expression qui est déterminée par l'ordre de l'évaluation qui fait parti de l'environnement.
La figure 3.6 représente schématiquement l'effet sémantique de l'évaluation d'une expression.

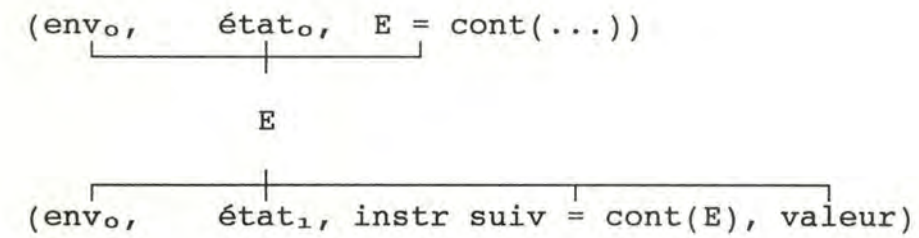


Figure 3.6 : l'effet de l'exécution d'une expression

On peut établir un parallélisme entre la composition syntaxique d'une commande et d'une expression. Une commande peut être constituée de sous-commandes et d'expressions. Une expression est une construction hiérarchique composée de sous-expressions. Si l'évaluation provoque un effet de bord, elle modifie aussi l'état de la mémoire. Par conséquent, une expression ne contient pas uniquement des sous-expressions mais aussi des commandes comme composants [TENNENT, R. D. p. 12].

La continuation d'une expression sera expliquée à l'aide de l'exemple *(k - l) + (j - i)*. On suppose que cette construction est évaluée de gauche à droite, *(k- l)* et *(j - i)* sont donc évalués dans l'ordre indiqué, avant de faire la somme.

Selon la définition d'une expression, l'évaluation de *(k - l)* fournit une valeur et un état. La continuation est déterminée par l'ordre de l'évaluation, par conséquent, il s'agit de *(j - i)*. Cette expression reçoit de la construction précédente, une valeur et un état. Elle préserve cette valeur, et l'expression est évaluée par rapport à l'état reçu. La continuation de *(j - i)* est la somme, elle est un opérateur binaire qui reçoit deux valeurs qui sont les résultats des sous-expressions.

La continuation de la somme reçoit la continuation de l'expression dans son ensemble, ce qui peut être une expression ou une commande. Celle-ci dépend de l'environnement. La continuation de l'expression recevra comme valeur, le résultat de la somme et comme état celui qui résulte de l'évaluation de la somme.

4.3.2 **Les effets de bord en C**

En toute généralité, dans les langages de programmation, un effet de bord peut par exemple apparaître quand une variable globale est modifiée dans un appel de fonction [GHEZZI & JAZAYERI p. 192].

Un effet de bord peut aussi être provoqué par une assignation qui se trouve parmi les opérateurs d'une expression. Ceci est souvent le cas en C. Par exemple, dans "*j = i++;*" *j* reçoit la valeur de *i*, mais en même temps *i* est incrémentée. Un autre exemple est "*Printf("%d\n", *--n);*". Avant l'affichage de **n*, le contenu de la variable *n* est décrémenté.

Le cadre théorique qu'on vient de définir, permet d'établir un lien entre les effets de bord et l'ordre de l'évaluation d'une expression.

Normalement, les composants d'une expression sont indépendants les uns des autres. Ceci implique que l'ordre dans lequel les sous-expressions sont évaluées, n'a pas d'effet sur la valeur produite par la construction dans son ensemble [GHEZZI & JAZAYERI p. 262].

Néanmoins, les effets de bord ne font pas toujours en sorte. Un composant peut modifier la valeur d'une variable qui est réemployée par un autre composant dans la même expression. Le résultat global dépendra alors de l'ordre de l'évaluation. En modifiant légèrement l'exemple précédent : *j = i + i++;*, on remarque tout de suite que l'ordre a une importance sur le résultat global de l'expression. Si *i++* est évalué avant *i*, alors la modification de l'état de la mémoire aura une répercution sur la valeur de *i* et sur le résultat global.

4.4 L'assignation comme expression et commande

4.4.1 Les L-values et les R-values

Une commande d'assignation est composée de deux expressions. Elle dispose de la forme $E_0 = E_1$. L'expression E_1 est évaluée est produit une R-value, c'est-à-dire, on accède au contenu d'un emplacement en mémoire. Cette valeur est stockée à la L-value ou l'adresse de E_0.

4.4.2 Le rôle du point-virgule

La sémantique de la commande d'assignation sera expliquée à l'aide de l'exemple *j = i += 5;*

On peut représenter *j = i += 5;* sous forme d'arbre, comme c'est illustré dans la figure 4.7.

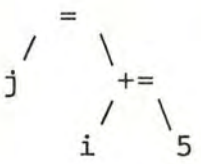


Figure 3.7 : représentation de *j = i += 5;* sous forme d'arbre

La construction *i += 5* est une expression avec un effet de bord. La valeur de l'expression est la valeur de l'opérande de droite après avoir effectué un cast vers le type de l'opérande de gauche. Concrètement la valeur de *i += 5* vaut "i + 5" car *i += 5* est équivalent à *i = i + 5*. Il y a un effet de bord, c'est-à-dire la valeur de *i* est incrémentée de 5.

Un autre exemple est *while ((x = getchar()) != EOF)*.

La assignation en C est donc une construction hybride qui est à la fois une commande et une expression. Elle est une expression quand la valeur qu'elle produit est utilisée dans l'évaluation d'une expression ultérieure, c'est-à-dire quand sa valeur est remontée dans l'arborescence. L'assignation est une commande quand la valeur qu'elle renvoie n'est plus employée. Quand une assignation est interprétée sémantiquement comme une commande, cela a un effet sur la syntaxe : la construction est terminée par un *;*.

4.5 Les procédures

Dans les langages de programmation, la construction de procédure permet de définir un nom pour un bloc d'instructions [GHEZZI & JAZAYERI p. 28].

Il existe deux types : la procédure d'expression (qui correspond sur le plan sémantique à une expression) et la procédure de commande (qui correspond sur le plan sémantique à une commande) [TENNENT, R. D. p. 20].

La définition d'une procédure fait partie de l'environnement. Elle peut être invoquée plusieurs fois par simple emploi de son nom.

Le langage C dispose uniquement de procédures d'expression. Néanmoins, beaucoup d'implémentations récentes prévoient le type "*void*" qui permet de définir une procédure de commande.

4.6 Les ruptures de séquence

4.6.1 Le contexte des ruptures de séquence

Il existe différents types de ruptures de séquence dans les langages de programmation.

Le "*goto label*" dispose d'une continuation explicite. Sémantiquement, l'exécution d'un goto a un effet statique. Sa destination est fixée par le programmeur. On peut même dire que le "*goto*" contient sa propre continuation, c'est-à-dire cette instruction n'a pas d'effet dynamique. L'instruction suivante est toujours celle dénotée par le "*label*", qui est un composant du "*goto*" [STOY J. p. 252]. Sur le plan mathématique, le "*goto*" correspond à une fonction constante qui pour n'importe quel argument (= la continuation) donne toujours le même résultat (= l'instruction dénotée par le label).

Il existe d'autres types de ruptures de séquence. Leur effet dépend du contexte dans lequel ils sont employés. La continuation des instructions "*return*", "*break*", "*continue*", "*exit*" dépendent de l'environnement dans lequel ces instructions sont exécutées. Ces constructions ont des continuations dynamiques [TENNENT, R. D. p. 150].

Remarque :
Si l'on veut être tout à fait rigoureux, la discussion précédente du rupture de séquence "*goto label*" est seulement applicable si le abel est une constante dont la valeur ne peut pas changer pendant l'exécution (comme dans le FORTRAN, le Cobol et le PL/1). Par contre, si le label est une variable dont la valeur peut changer, "*goto label*" a un effet dynamique [TENNENT, R. D. p. 150].

4.6.2 Le langage C

L'instruction de rupture de séquence avec une destination explicite est le *goto*.

Les instructions de rupture de séquence avec une destination implicites sont le *break*, le *continue* et l'*exit()*.

Chapitre IV :

Des aspects de la performance du langage C

Les aspects syntaxiques et sémantiques du langage C ont été traités dans les chapitres précédents. Cette partie du mémoire présente les résultats d'un ensemble de tests effectués sur le C.

Il y a deux grands volets dans ce chapitre. Ils reflètent les deux aspects du langage C qui ont été soumis à des tests. D'un côté, les temps d'exécution des instructions individuelles ont été mesurés sur différents types de machines. D'un autre côté, des programmes en C dans leur ensemble ont été soumis à des essais. Avant d'aborder la description des tests et les considérations sur les résultats, quelques généralités sur l'efficacité d'un langage en général seront évoquées.

Ce chapitre est basé sur un résumé de deux travaux effectués dans le cadre du cours de "mesures et performances" appartenant au programme de la deuxième licence et maîtrise informatique du F.N.D.P. . Il s'agit plus particulièrement d'un travail effectué par Filip Perquy et Bernard Thiry sur les performances des instructions en C : "*Rapport sur Mesures et Performances*", et d'un travail fourni par Jean-Marc Albrecq, Jean-Marie Belle, Philippe Cayphas, Bernard Decuyper, Etienne Degrady et Françoise Dubisy sur l'efficacité en temps d'exécution et de compilation de programmes en C sur les systèmes d'exploitation UNIX et VMS : "*Mesures et Performances : Comparaison des performances d'un programme C sur deux OS différents : UNIX Version 4.2 et mini VMS*".

1 Généralités

1.1 Des considérations sur l'efficacité d'un langage

Dans l'informatique, le terme "efficacité d'un langage" couvre plusieurs notions.

Primo, ce terme évoque la notion de place occupée sur un certain support (ex : sur disque ou en mémoire centrale). Dans cette optique, il vaut mieux de parler de **compacité** du code source, objet ou exécutable.

Secundo, l'efficacité implique un élément de **temps**. A ce point, il faut distinguer entre le temps de préparation d'un programme (ou plus précisément le temps nécessaire pour effectuer le codage du programme jusqu'à la compilation sans erreurs) et le temps d'exécution. Ce temps d'exécution couvre à nouveau deux notions. Il faut distinguer entre le temps d'exécution d'un programme entier et le temps d'exécution des instructions individuelles.

Le temps de préparation d'un programme est surtout influencé par l'environnement dans lequel sont faits le codage et la compilation. La qualité des éditeurs et de la documentation qui sont disponibles jouent des rôles déterminants. Les éléments du langage intervenant à ce stade, sont en premier lieu des aspects syntaxiques. Le langage facilite-il l'écriture d'un programme lisible ou pas ? Le langage C permet d'écrire des programmes très compacts. Un exemple de cet aspect est une fonction qui permet de copier une chaîne de caractères dans une autre.

```
strcpy(x, y)
char *x, *y;
{
     while(*x++ = *y++) ;
}
```

Exemple 4.1 : Version compacte d'un programme de copie de chaînes de caractères

[KERNIGHAN B. W. & RITCHIE D. M. p. 98].

Dans la partie test de la boucle, ce programme effectue deux adressages indirects via les pointeurs, une affectation, le test de fin et ensuite éventuellement deux incrémentations. Tout se fait dans une ligne de programme. Ce style de codage peut facilement provoquer des erreurs très difficiles à déceler.

Le programme précédent offre un exemple d'une philosophie générale en C. Les expressions et les programmes peuvent être écrites de façon très compacte. Mais finalement le programmeur décide lui-même du niveau de compacité. Il n'est pas obligé de programmer de cette façon. L'exemple 4.2 reprend le même programme, mais cette fois-ci, le style adopté est moins dense.

```
strcpy(x, y)
char x[], y[];
{
     int i, j;
     i = 0;
     j = 0;

     while(y[j] != '\0') {
          x[i] = y[j];
          i++;
          j++;
     }
}
```

Exemple 4.2 : Version plus élaborée de d'un programme de copie de chaînes de caractères

En C, le programmeur peut utiliser le préprocesseur pour contourner les difficultés syntaxiques. A l'aide de cet outil, il peut par exemple remplacer *&&* et ¦¦ par *AND* et *OR*, *&* et ¦ par *BAND* et *BOR* (B indique binair), etc. Il peut ainsi construire ses propres définitions et via le "*#include <fichier>*", il peut les appeler et les rendre exécutables.

Les aspects du concept "efficacité", bien qu'importants, ne seront pas considérés en détail dans ce chapitre. Par contre, on développera le concept d'efficacité dans le sens de temps d'exécution d'un programme entier et des instructions individuelles en C.

1.2 Domaine des tests et cadre de travail

L'évaluation des performances des instructions individuelles constitue un premier domaine de tests. Ces tests ont d'abord été effectués sur le PDP-11 et cela pour des raisons historiques, la première implémentation commercialisée du langage C ayant été développée sur cette machine. Les essais auront pour but de déterminer s'il existe une différence en termes de temps d'exécution entre les instructions typiques pour le C et les formes classiques. Ensuite, les mêmes tests seront effectués sur des VAX 750 (sous UNIX) et cela pour contrôler si l'on retrouve les mêmes tendances.

L'efficacité d'exécution d'un programme entier dans son environnement constitue un deuxième domaine de tests. Deux systèmes d'exploitation seront ainsi confrontés l'un à l'autre. Il s'agit de UNIX et de VMS.

L'ensemble des machines sur lesquelles les tests ont été faits sont disponibles à l'Institut d'Informatique, et accessibles via le PABX. Il s'agit de :

- "CIRCE", un PDP-11/84 opérant sous UNIX version V7,
- "JUNON", un VAX 750 opérant sous UNIX BSD 4.2 (avec quelques drivers de la version 4.3),
- "ALMA", un VAX 750 géré sous ULTRIX 32 version 1.,
- "DIANE", un autre VAX 750 travaillant sous VMS 4.7,
- "VENUS", qui est aussi un VAX mais de type 8600 opérant sous VMS 4.7.

2 Tests sur les performances des instructions individuelles

La section "Tests sur les performances des instructions individuelles" interprète les temps d'exécution de certaines instructions de calcul numérique en C.

Cette section est composée de trois grandes parties. D'abord les hypothèses de travail et le type des instructions testées sont présentées. Ensuite, le domaine des essais avec les instructions en C soumises aux tests est expliqué.

2.1 Hypothèses de travail

Le langage C dispose d'une large gamme d'opérateurs, une trentaine au total. Parmi ces opérateurs, on a l'impression qu'il y en a beaucoup qui font double emploi. Il s'agit plus particulièrement des opérateurs d'incrémentation, de multiplication et de shift (par exemple : .+=. ou .*=. par rapport à .=.+. ou .=.*.). Les concepteurs du langage, Kernighan et Ritchie restent eux-mêmes assez vagues sur l'origine de cette redondance.
Selon Kernighan et Ritchie, elle serait justifiée à cause de différence dans la vitesse d'exécution de ces instructions :

- .++ est décrite comme une opération plus concise et souvent plus efficace [KERNIGHAN B. W. & RITCHIE D. M. p. 19].
- L'opérateur .+=. ne provoque qu'une seule évaluation de l'opérande de gauche [KERNIGHAN B. W. & RITCHIE D. M. p. 46].

Ces affirmations forment une base quant aux tests à effectuer. On a voulu vérifier, en général, si l'implémentation des instructions propres au C est plus performante que leurs contreparties traditionnelles.
La hiérarchie des opérateurs suggérée par Kernighan et Ritchie est-elle respectée pour l'addition et la multiplication, c'est-à-dire que serait .++ plus performant que .+=. qui à son tour serait plus performant que .=.+. pour l'addition et .*=. plus rapide que .=.*. pour la multiplication ?

En même temps, l'opérateur de décalage a été testé. Il existe sous deux formes syntaxiques : ".<<." et ".>>.". Le premier, ".<<." permet de décaler les bits vers la gauche, ce qui peut correspondre à une multiplication entière de puissance de 2. Dans l'exemple *x<<2;*, *x* est multiplié par 4 (2^2). Le second opérateur ".>>." décale les bits vers la droite, ce qui correspond à une division.

Dans le cadre des tests, seul le décalage vers la gauche est repris. Comme il correspond à la multiplication par 2^n, cet aspect des tests permet de vérifier dans quelle mesure le compilateur est capable de remplacer une multiplication par 2^n par un shift de n bits vers la gauche.

2.2 Domaine des tests

2.2.1 **Les additions**

Les tests suivants ont été effectués sur les trois types d'opérateurs d'addition :

- pour les entiers : k++, k+=1, k=k+1, k+=0, k=k+0, k+=13 et k=k+13 (où k représente une variable entière),
- pour les réels : r++, r+=1 et r+=1., r=r+1 et r=r+1., r+=0 et r+=0., r=r+0 et r=r+0., r+=13 et r+=13., r=r+13 et r=r+13. (où r désigne une variable réelle).

La première série de tests consiste en une comparaison des temps d'exécution pour l'addition entière. Les comparaisons de k++ avec k+=1 et k=k+1, et de k+=1 et k=k+1 par rapport à k+=13 et k=k+13 (addition d'un nombre entier quelconque) ont formés les tests de base. On a ajouté les opérations k+=0 et k=k+0 pour vérifier si le compilateur détecte qu'il s'agit d'une addition avec 0, (zéro) élément neutre pour l'addition.

La deuxième série consiste en des essais avec des opérandes réels. Mais on a aussi testé des opérations mixtes (i.e. là où cela était possible, on a testé à la fois une addition avec des opérandes uniquement réels et une addition avec une des opérandes entiers). Ceci a permis de tester si le temps de la conversion d'un entier en un réel prêtait à interférence. Ici aussi on retrouve l'addition avec zéro pour les mêmes raisons que précédemment.

2.2.2 Les multiplications

Deux autres types d'opérateurs soumis aux tests sont la multiplication et l'opérateur de décalage.

Comme pour les additions, on a fait la distinction entre les entiers et les réels.
Pour les entiers, les essais suivants ont été faits :
k*=0, k=k*0, k*=1, k=k*1, k*=2, k=k*2, k<<0, k<<=0, k=k<<0, k<<1, k<<=1, k=k<<1, k*=13, k=k*13, k*=0., k=k*0., k*=1., k=k*1., k*=2., k=k*2., k*=13., k=k*13.
Et pour les réels :
r*=0, r=r*0, r*=1, r=r*1, r*=2, r=r*2, r*=05, r=r*05, r*=0., r=r*0., r*=1., r=r*1., r*=2., r=r*2., r*=05., r=r*05.

Ces instructions ont été groupées de différentes façons afin de découvrir des correspondances et des interrelations.

D'abord, on a groupé tout ce qui est multiplication entière c'est-à-dire : k*=0 et k=k*0, k*=1 et k=k*1, k*=2 et k=k*2, k*=13 et k=k*13. Parmi ces entiers se trouvent deux éléments particuliers "**0**" qui est l'élément neutre pour la multiplication et "**1**" l'élément neutre.

Un deuxième ensemble est formé par les multiplications entières mixtes (avec un opérande réel), le résultat étant stocké dans une variable entière : k*=0. et k=k*0., k*=1.et k=k*1., k*=2. et k=k*2., k*=13. et k=k*13. .

Puis on retrouve les multiplications réelles mixtes (avec un entier) mais dont le résultat est stocké dans une variable réelle. Il s'agit des instructions suivantes : r*=0, r=r*0, r*=1, r=r*1, r*=2, r=r*2, r*=05, r=r*05 .

Les multiplications réelles uniquement composées d'opérandes réels sont regroupées dans la comparaison suivante : il s'agit de r*=0., r=r*0., r*=1., r=r*1., r*=2., r=r*2., r*=05., r=r*05.

Ensuite, les comparaisons entre l'opération de décalage et la multiplication ont été développées.

2.3 Interprétation des résultats

2.3.1 **Présentation des tableaux**

Les sections précédentes ont présenté les hypothèses de travail et le domaine des tests.

Les résultats de ces tests se trouvent dans l'annexe A, dans la partie "2.2 Les tableaux avec les instructions individuelles". Chaque tableau contient les résultats d'une session de prise de mesures. Plus concrètement, il s'agit des tableaux portants les noms de "Alma 11 (= 11 avril 1988)", "Alma 12 (= 12 avril 1988)", "Circé 08 (= 8 avril 1988)", "Circé 11 (= 11 avril 1988)", "Junon 11 (= 11 avril 1988)" et "Junon 12 (= 12 avril 1988)". Ces noms indiquent à la fois la machine testée et la date à laquelle les tests se sont effectués.

Chaque tableau est construit selon la même structure de base. Il est composé de deux parties.

Primo, il y a le tableau en lui-même contenant les temps d'exécution des diverses instructions, qui se trouve dans la partie supérieure. Chaque instruction mentionnée dans la colonne de gauche a été exécutée 10.000.000 fois (la valeur entre [] indique la valeur de l'initialisation de la variable).

Pour obtenir le temps d'exécution, l'instruction à mesurer a été placée dans une boucle. Cette boucle a permis de mesurer deux types de temps : d'une part le temps d'exécution de la boucle avec l'instruction comprise ("la boucle remplie") et d'autre part le temps de la boucle sans l'instruction exécution ("la boucle vide"). La soustraction de ces deux temps ("la boucle remplie" - "la boucle vide") donne une sorte de temps réel nécessaire pour exécuter l'instruction seule. Dans le tableau, le temps réel se trouve dans la colonne de droite. Une description détaillée de la prise de mesures et le traitement des résultats pour obtenir ce temps réel se trouvent dans l'annexe A dans la section 2.1.

Secundo, les temps d'exécution sont représentés sous forme de graphe dans la partie suivante.

La structure de la section "2.3 Interprétation des résultats" a été conçue telle qu'elle permet la vérification systématique des hypothèses. D'abord, les résultats de l'addition entière et réelle seront présentés. Ensuite, les différents temps obtenus pour divers types de multiplication seront interprétés (multiplication entière, multiplication entière avec opérande réel, multiplication réelle avec opérande entier, multiplication réelle, multiplication par rapport à l'opérateur de décalage). L'indication dans les titres "cas A, B, C, D, F et G" correspondent à la répartition des tableaux dans l'annexe A dans la section 2.2.1 jusque 2.2.6.

Les résultats de Circé sont d'abord envisagés. En effet, Circé est la première machine sur laquelle le langage C a été développé et commercialisé. Celle-ci constitue la machine de référence pour les tests.

Ensuite, les résultats recueillis sur Junon et Alma sont présentés. Les différents temps d'exécution obtenus sur ces deux machines sont très proche un de l'autre, ainsi on a pu les considérer ensemble.

2.3.2 **Circé**

2.3.2.1 Les additions

<u>L'addition entière - cas "A"</u>
Une constatation rapide des tableaux "Circé 08" et "Circé 11" révèle que l'instruction k+=0 s'est exécutée 10.000.000 fois en 0,08 sec et ces 8 centièmes sont encore dus à des erreurs d'arrondis provoquées par les calculs sur les résultats ! Seule une instruction qui ne fait vraiment rien peut arriver à un tel résultat. k+=0 est sans doute détectée et automatiquement traitée comme une opération vide ou bien un NOP ("no operation").

D'autres constatations évidentes :
- l'instruction k+=. est plus performante que k=k+.
- 10.000.000 fois k++ donne le même temps que 10.000.000 k+=1. Ces deux instructions sont probablement implémentées de la même façon.
- Entre k+=1 et k=k+1 existe une différence de 11 sec, différence que l'on retrouve entre k+=13 et k=k+13 .

L'addition réelle - cas "B"

L'addition réelle se comporte différemment par rapport aux hypothèses posées.
On peut distinguer trois groupes dans les temps d'exécution,
1) r++
2) r=r+1, r=r+1., r=r+0, r=r+0. r=r+13 et r=r+13.
3) r+=1, r+=1., r+=0, r+=0., r+=13 et r+=13.

On constate que l'implémentation de l'opérateur assignation-addition est presque deux fois moins rapide que l'opération d'addition classique suivie de l'assignation (r=r+.).

Il est à noter que l'implémentation de l'addition réelle ne reconnaît pas véritablement le zéro (l'élément neutre). r=r+0 est quand même plus performant que r=r+1, mais ceci est sans doute dû au fait que dans le cas d'une addition avec zéro, il n'y a aucun travail qui doit être fait par l'accumulateur afin de normaliser le nombre (il ne doit pas décaler de bits).

Ici aussi les constantes entières sont considérées comme des réels : pas de temps de conversion d'un entier vers un réel. Il y a, néanmoins, une exception, apparemment inexplicable, dans le cas de r+=1. . Cette instruction nécessite pour une quelconque raison, besoin de 18 sec en plus que toutes les autres !

Une comparaison entre l'addition entière et réelle
L'addition entière est plus rapide que l'addition réelle ce qui est normal. On obtient deux résultats étonnants. Le r++ est très performant mais reste quand même moins rapide que l'addition entière la plus lente. L'addition du type r=r+. est plus performante que r+=. L'hypothèse que .+=. est plus performant que .=.+. n'est donc pas vérifiées dans tous les cas.

2.3.2.2 Les multiplications

La multiplication entière - cas "A"
L'instruction k*=1 se fait en 0 sec. Comme dans le cas de l'addition, l'élément neutre (le 1) est détecté. Sans doute, cette instruction est aussi remplacée par un NOP.

Les temps d'exécution de la multiplication entière peuvent être répartis en quatre groupes :
1) k*=1
2) k=k*0 et k*=2
3) k=k*1 et k=k*2
4) k*=0, k*=13 et k=k*13

Les relations entre k*=. et k=k*. d'un côté et son opérande de l'autre côté sont représentées dans la table 4.1. Comme cette table l'indique, il est difficile de dire laquelle des deux formes de multiplication est la plus performante. L'opération k=k*. est plus rapide que k*=. pour les opérandes 1 et 2. Par contre, k*=. est plus rapide pour 0. Utiliser k=k*. ou k*=. avec l'opérande 13 résulte dans le même temps d'exécution.

Table de comparaison :

opérande	k*=. par rapport à k*=.*.
0	>
1	<
2	<
13	=

Table 4.1 : représentation des tendances dans les temps d'exécution des opérandes pour l'addition réelle

La multiplication entière avec opérande réel - cas "B"
L'utilisation de l'opérateur d'assignation-multiplication, dans le cas de la multiplication entière avec opérande réel est moins performant que l'opération classique.
L'instruction k=k*. est presque deux fois plus rapide que k*=. .

Cette constatation est basée sur les deux grands groupes qu'on peut distinguer dans les temps d'exécution :
1) k=k*0., k=k*2., k=k*13. et k=k*1.,
2) k*=0., k*=1., k*=2. et k=k*13. .

La multiplication réelle avec opérande entier - cas "C"
Pour la multiplication réelle avec opérande entier, les graphes indiquent deux groupes distincts dans les temps :
1) r=r*0, r=r*1, r=r*2 et r=r*05
2) r*=0, r*=1, r*=2 et r*=05 .

Comme dans le cas de la multiplication entière avec opérande réel, on retrouve les mêmes tendances : r*=. est moins performant que r=.*.

On retrouve deux instructions "type" à la base de chaque groupe, il s'agit de r=r*. pour le premier groupe et r*=. pour le deuxième.

La multiplication par zéro est la plus performante dans les deux classes, l'opérande nul est traité à l'intérieur de la multiplication et détecté comme tel, certaines opérations ne seront pas effectuées. Mais ceci est à différencier de la véritable détection du zéro, dans le cas de k=k*0 (cfr le cas "A").

La multiplication réelle - cas "D"
A nouveau, on rencontre les mêmes tendances : r=r*. est plus rapide que r*=.

Les opérateurs de décalage et la multiplication - cas "F" et cas "G"

La comparaison entre les opérateurs de décalage et les opérateurs de multiplication montrent que k*=2 est égal à k<<=1 et que k=k*2 est égal à k=k<<1 en temps d'exécution. La multiplication par 2 est donc bien détectée et remplacée par un décalage de 1 vers la gauche.

Comme dans le cas k+=0, un opérande nul a été détecté dans les instructions k<<0 et k<<=0. Les deux temps d'exécution correspondants sont nuls.

Remarquez que les instructions k<<1 et k<<=1 donnent un temps d'exécution différent, bien que le résultat final des deux instructions est identique. La construction k<<1 est une expression avec effet de bord, l'état de la mémoire est modifié. Par contre, k<<=1 est commande.

2.3.2.3 Une coïncidence particulière

Le langage C place les instructions du type .=., .+=., .*=. et .<<=. au même niveau sémantique. Dans tous les cas, il s'agit d'un opérateur d'assignation. L'étude des temps d'exécution révèle qu'on retrouve cette même équivalence pour certains opérandes entiers : exécuter 10.000.000 de fois k=1, k+=1, k*=2 et k<<=1 demande chaque fois presque 17,92 sec et cela dans deux sessions de prises de mesures différentes (Circé 08 et Circé 11).

2.3.3 **Alma et Junon**

Les machines ALMA et Junon sont dans l'ensemble, très semblables. Dès lors, on a interprété ensemble les résultats récoltés sur ces deux machines.

2.3.3.1 Les additions

Addition entière - Cas "A"

Lors des premières constations, on peut séparer les temps en deux classes : d'un côté k++, k+=1, k+=0 et k+=13 et de l'autre côté, k=k+1, k=k+0 et k=k+13 . On retrouve donc les mêmes tendances que sur Circé.

Une des questions qu'on peut alors se poser est la suivante : pourquoi k+=. est-il plus performant que k=k+. ?
La différence entre k+=. et k=k+. se trouve dans le fait qu'il y a une initialisation en plus. Donc, d'un point de vue strictement temporel, k=k+. doit plus ou moins être égal à la somme du temps nécessaire à k+=. et du temps d'une initialisation (10.000.000 fois k=0 se fait en ± 21 sec).

Les résultats n'infirment pas cette hypothèse. k=k+1 est bien une combinaison d'une opération d'addition entière plus une initialisation.

Addition réelle - Cas "B"

L'interprétation des résultats révèle trois groupes de temps qui correspondent aux instructions suivantes :

1) r++,
2) r+=0, r+=0., r=r+0 et r=r+0.,
3) r+=1, r+=1., r=r+1, r=r+1., r+=13, r+=13., r=r+13 et r=r+13. .

L'égalité en temps d'exécution entre les instructions k++ et k+=1 n'existe plus au niveau des réels : r++ est nettement plus rapide r+=1 .

L'addition réelle avec une constante entière (r+=0, r=r+0 et r+=13) est chaque fois légèrement plus rapide que l'addition réelle avec un réel (r+=0., r=r+0. et r+=13.). Comment peut-on expliquer cette différence ?

Son origine doit probablement être cherchée au niveau assembleur. Pour les opérations en virgule flottante, les instructions ont la forme <mne><cst>, où <mne> est une abréviation pour une instruction mnémonique et <cst> pour une constante. En hexadécimal, <cst> peut représenter soit une

adresse, soit une valeur entière en base 16. Les nombres en virgule flottante sont en général représentés via une adresse tandis que les entiers sont directement accessibles par une constante. Les réels sont donc obligatoirement placés en zone de données, les entiers pas, ils peuvent directement être manipulés par le programme.

Par conséquent, il doit y avoir une relation entre les opérandes 0 et 0., entre 1 et 1. et entre 13 et 13. La différence se trouve dans un accès mémoire en plus, ce qui doit se confirmer dans les temps d'exécution. Effectivement, on remarque chaque fois une différence aux environs de 4 secondes entre une addition réelle avec opérande entier par rapport à une une addition entière avec opérande réel.

2.3.3.2 Multiplication

La multiplication entière - cas "A"

Cinq groupes de temps sont à distinguer : k=k*1 < k=k*2 < k*=0, k*=1 et k*=2 < k*=13 < k=k*0 et k=k*13 .

Les relations entre k*=. et k=k*. d'un côté et son opérande de l'autre côté sont représentées dans la table 4.2. Cette table indique que dans tous les cas (sauf la multiplication avec 0) k=k*. est plus performant que k*=. . Cette constatation est à nouveau contraire à l'hypothèse posée, c'est-à-dire la multiplication .*=. est plus performante que .=.*. .

opérande	k*=. par rapport à k=k*.
0	<
1	>
2	>
13	>

Table 4.2 : représentation des tendances dans les temps d'exécution des opérandes pour la multiplication entière

La multiplication entière avec opérande réel - cas "B"
L'analyse des graphes révèlent que l'instruction k*=. s'exécute légèrement plus rapidement que k=k*. . La multiplication entière avec opérande réel a des tendances d'exécution différentes de la multiplication entière.

En plus les temps d'exécution de l'instruction k*=. et k=k*. pour l'opérande 1. et 13. sont très élevés sur la machine Alma et cela pour les deux sessions de test. Junon présente le même phénomène pour l'instruction k=k*. combinée avec l'opérande 1. et 13. . Apparemment la conversion vers un entier pose des problèmes.

La multiplication réelle - cas "C" et "D"
Les temps obtenus pour la multiplication réelle mixte (multiplication réelle avec un entier) sont comparables aux temps obtenus pour la multiplication réelle pure.

L'instruction r*=. est chaque fois légèrement plus rapide (± 3 secondes) que r=r*. et cela par rapport à un temps d'exécution de 490 secondes en moyenne pour 10.000.000 d'instructions. Cette constatation permet de dire que pour une exécution normale d'un programme, l'emploi de r*=. ou r=r*. aboutit au même temps d'exécution.

L'opérateur de décalage - cas "F" et cas "G"
Les comparaisons entre l'opérateur de décalage et la multiplication permettent de vérifier dans quel mesure le compilateur est capable de remplacer une multiplication de 2^n par un décalage ?

Il y a bien remplacement de la multiplication par 2 par un shift de 1 (en effet, sur 10M d'instruction on retrouve chaque fois un temps de 72 ± secondes). Mais ce qui est étonnant, c'est que la multiplication classique, type k=k*., reçoit ce régime de faveur, la multiplication-assignation propre au C, doit-elle se contenter d'une multiplication non optimisée ?

2.3.4 Les conclusions

La discussion sur les différents temps démontrent clairement qu'il est impossible de confirmer les hypothèses posées au début de ce chapitre. On peut même constater des résultats contraires aux hypothèses posées.

La hiérarchie .++ < .+=. < .=.+. est vérifiée pour des opérandes entières. En ce qui concerne les opérandes réelles, r++ est incontestablement plus rapide que r+=1. La différence entre .+=. et .=.+. pour les entiers est beaucoup moins prononcée pour les réels.

La considération des temps d'exécution des deux opérateurs de multiplication dans leur ensemble montrent qu'il est difficile de déterminer si .=*. ou .=.*. est le plus performant. En plus, les temps de conversion modifient le temps d'exécution de l'instruction pure.

Ces tests démontrent aussi qu'il ne faut pas uniquement considérer les instructions individuellement, mais avec leurs opérandes. Plus généralement, il faut étudier une instruction dans son environnement. Le deuxième volet de ce chapitre étudiera l'influence de l'environnement d'un programme sur le temps d'exécution.

Quelques considérations importantes sont :
- Avant tout, saluer la bonne performance générale de Circé. Peut-être est-ce dû à une utilisation particulière, plus orientée "mono-utilisateur". En effet, les processus de test se sont retrouvés souvent seuls à s'exécuter.
- Alma et Junon sont de véritables soeur jumelles : résultats presque identiques, même dépendance vis-à-vis de la charge et mauvais comportements pour chacune des machines. (Junon est un rien plus dégourdie que Alma mais les différences entre les résultats obtenus sont faibles).

MAIS il faut néanmoins relativiser tous les résultats obtenus. Voici un benchmark décrit dans la série "Applications Notes" de chez Intel sur le processeur numérique 8087 [INTEL, p. S-3] :

instruction	temps approximatif d'exécution (µsec) 5 MHz clock	
	8087	8086-8088
Mul. (simple précision)	19	1600
Mul. (double précision)	27	2100
Addition	17	1600
Division simple précision)	39	3200
Comparaison	9	1300
Chargement d'une variable	9	1700
Mémorisation d'une constante	18	1200
Racine carrée	36	19600
Tangente	90	13000
Exponentiation	100	17100

Table 4.3 : les temps d'exécution d'une instruction de calcul numérique sur le processeur 8087

Ce tableau donne une comparaison des temps d'exécution de certaines instructions exécutées sur PC-compatibles avec et sans co-processeur de calcul.

On y découvre entre autres des résultats de 2100 µsec pour une multiplication en double précision sans co-processeur de calcul et un temps de **27** µsec avec co-processeur. Les temps réels moyens, pour une multiplication, obtenus sur Circé, Alma, Junon sont respectivement et approximativement **9.38** µsec, **27.3** µsec et **25.8** µsec. Ces résultats ne rehaussent pas les performance des trois ordinateurs face à ceux obtenus sur un micro-ordinateur bien équipé (dans le domaine du calcul numérique) !

3 Tests sur les performances des programmes complets

3.1 Objectif des tests

Les essais menés dans la deuxième partie de ce chapitre ont comme but de tester l'interaction d'un programme C avec son environnement, c'est-à-dire le système d'exploitation.

Les performances d'un programme C sont comparées sur deux systèmes d'exploitation différents : Unix Version 4.2 et mini VMS. Les programmes d'essais qui se trouvent en annexe, n'ont pas d'orientation particulière. Il s'agit de programmes testant la vitesse des boucles "while" et "for", l'influence de la déclaration "register", la vitesse de l'écriture à l'écran, d'accès aux fichiers, d'un appel de fonction et du calcul numérique.

Les programmes sources, une description de la prise des mesures et les mesures elles-mêmes se trouvent en annexe dans la section "Tests sur les performances des programmes complets".

3.2 Interprétation des résultats

Quand on compare d'une manière générale les valeurs dans les colonnes "Unix Exec" et "VMS Exec" du tableau "Tableau récapitulatif des moyennes", on constate que l'exécution des programmes sur VMS semble plus rapide que sur UNIX, à l'exception des accès aux fichiers.

Ces constatations semblent contraires à une certaine réalité. Le C évoque généralement le nom de Unix. Les deux ont la réputation d'être efficace. Les mesures qu'on vient d'étudier contredisent cette hypothèse. Comment peut-on expliquer les causes des performances de C sous VMS ?

Le temps de compilation sur VMS est en moyenne sept fois plus élevé que le temps de compilation sur Unix. La consultation de "Tableau récapitulatif des tailles" révèle que les versions exécutables sous VMS sont environ six fois plus

longues que celles de Unix. Le temps de compilation serait donc lié à la création, par le compilateur d'un code exécutable plus long.

Une explication possible serait que le compilateur sous VMS cherche à optimiser le code exécutable. Une façon d'optimiser le code serait de recopier dans le code exécutable l'entièreté (ou des parties) de librairies auxquelles le programme fera appel lors de son exécution. Ceci expliquerait le fait que les codes exécutables sur VMS ont une taille relativement uniforme proche de 38.000 bytes. Cela serait dû au chargement de la même librairie lors de la compilation.

4 Les conclusions

Dans ce chapitre, les résultats de différents tests sur le langage C ont été présentés.

Au niveau des instructions individuelles, il est difficile sinon impossible de dire que la hiérarchie suggérée par Kernighan et Ritchie, est vérifiée dans la réalité.

Le deuxième volet de ce chapitre a démontré que l'environnement dont un programme dispose, notamment le système d'exploitation, se révèle comme décisif pour la vitesse exécution d'un programme.

Les conclusions tirées de ces différents tests indiquent qu'il faut plutôt penser à optimiser un programme complet. Les interrelations entre les différents composants d'un programme doivent être étudiées. En fonction de ces résultats, les morceaux d'un programme les plus coûteux en temps d'exécution doivent être optimisés, éventuellement en les codant en assembleur et en faisant appel à des processeurs spécialisés (cfr la comparaison entre un 8087 pour un micro-ordinateur et un VAX 750 au sujet du calcul numérique).

Conclusion

Dans ce mémoire, une approche globale du langage C a été développée.
Aprés une brève présentation d'une typologie des langages de programmation, les diverses constructions syntaxiques du langage C ont été envisagées.

Ensuite, les principes dénotationnels ont permis d'introduire un modèle sémantique. La notion de contexte d'une instruction (environnement, état de la mémoire et continuation) constitue les piliers de ce modèle. Cette notion a fourni une base qui a permis de correctement interpréter les constructions syntaxiques du C.

Dans l'introduction de ce mémoire, les qualités importantes du langage C ont été mentionnées. Les aspects de performance ont été testés dans le chapitre IV. Les différents essais indiquent qu'une instruction individuelle doit être considérée en combinaison avec son opérande et un programme avec son système d'exploitation afin de tirer des conclusions significatives. En toute généralité, ces tests ont montré que l'efficacité du langage C est fort lié à l'environnement d'exécution.

En même temps, cette étude a permis au rédacteur de ce mémoire d'approfondir sa formation comme informaticien dans le domaine des langages de programmation en général et le langage C en particulier.

Références

ALBRECQ Jean-Marc, BELLE Jean-Marie, CAYPHAS Philippe, DECUYPER Bernard, DEGRADY Etienne et DUBISY Françoise "Mesures et Performances : Comparaison des performaces d'un programme C sur deux OS différents : UNIX Version 4.2 et mini VMS", F.N.D.P., 1987, 39 p.

ANDERSON, "Type syntax in the language C", SIGPLAN Notices, vol 15, n° 3, p. 21-27.

BACKUS John, "Can Programming be liberated from the Von Neumann Style ? A Functionnal Style and Its Algebra of Programs", Commun. of the ACM, August 78 (12), p. 613-641.

BERNARD Thiry & PERQUY Filip, "Rapport sur Mesures et Performances", F.N.D.P., 1988, 282 p.

BOURNE, "The UNIX System", Addison Wesley, 1982, 300 p.

CLINGER W. "The Semantics of Scheme", Byte, McGraw-Hill, N. Y., February 1988, p. 221-227.

DIGITAL EQUIPEMENT CORPORATION, "The PDP-11", DIGITAL, 1983, 272 p.

FEUER & GEHANI, "Ada, C and PASCAL" paru chez Prentice Hall, Englewood Cliffs, N. J., 271 p. .

FITZHORN & JOHNSON, "C : Towards A Concise Syntactic Description", SIGPLAN Notices, vol 16, n° 12, p. 14-22.

JENSEN & WIRTH, "Pascal User Manual and Report", New York, Springer Verlag, 1975, 165 p.

HOARE C. A. R. "Prospects for a better Programming Language", High Level Languages, Infotech State of art report 7, England, 1972, p. 328-343.

INTEL, "Application notes", Santa Clara, 1981, p.S1-S88.

KERNIGHAN B. W. & RITCHIE D. M., "Le langage C", Masson, Paris, 1987, 218 p.

KNUTH, D. E., "Structured programming with goto statements", Computing Surveys, n° 6, 1974, p. 261-301.

LANDIN P., "The next 700 Programming Languages", Comm. of the ACM, 1966, vol 9 (3), p. 157-164.

LE ROBERT, "Le Micro Robert", Paris, 1971, 1175 p.

MARCOTTY Michael & LEDGARD Henry, "The world of Programming languages", Springer-Verlag, N.Y., 360 p.

MATETI P., "Pascal versus C : A subjective Compararison" in Comparing and Accessing Programming Languages : Ada, C and PASCAL, Prentice Hall, Englewood Cliffs, N. J., 271 p.

MEINADIER Jean-Pierre, "Structure et fonctionnement des ordinateurs", Larousse, Paris, 1971, 401 p.

RITCHIE D. M., "The C programming Language - Reference Manual", Bell Laboratoires, Murray Hill, N.J., 31 p.

SAMMET, "Programming Languages, History and Fundamentals", Printice-Hall, Englewood-Cliffs N.J., 785p.

STOY J. E., "Denotation Semantics : The Scott-Strachey Approach to Programming Langugge Theory", The MIT Press, Cambridge, 1977, 413 p.

STRACHEY C., "Towards a formal semantics" (dans Formal Language Description Languages for Computer Programming, Ed.

STEEL T. B., North-Holland Publishing Company, Amsterdam, 1971, p. 198-220.

TENNENT R, "Principles of Programming Languages", Prentice Hall, N.J., 251 p.

"The Bell System Technical Journal", Bell Laboratories, Murray Hill, New Jersey, July-August, 1978, n° 6.

TOMBE Karl. "Le petit guide du langage C", Centre de Recherche en Informatique de Nancy, Vandæuvre-les-Nancy, 1986, 29 p.

VAN LAMSWEERDE Axel , "Le cycle de vie d'un projet de développement d'un logiciel", F.N.D.P., Namur, p. 23-73.

Annexes

1 Introduction

La partie annexe de ce mémoire est composée de deux volets :
- la partie A qui comprend les tests sur les instructions individuelles,
- la partie B qui comprend les tests sur les performances des programmes complets.

Chaque partie est basée sur la même structure :
- méthode de prise de mesures et description des programmes,
- les tableaux contenant les résultats,
- les programmes sources.

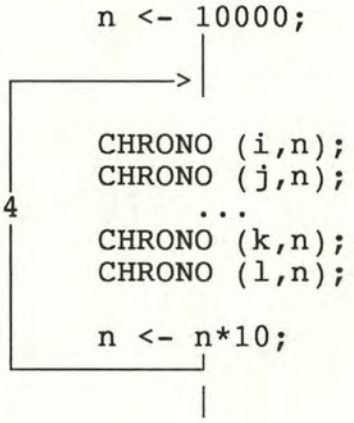
2 Les tests sur les instructions individuelles

2.1 Méthode de prise de mesures

2.1.1 Structure générale (schéma)

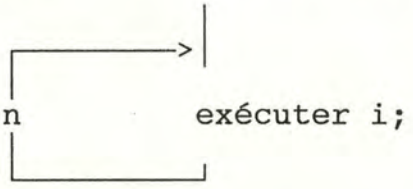
Chacun des programmes générant les résultats des tests a été conçu sur le même schéma. Un ensemble de chronométrages d'instructions, eux-même, placés dans une boucle, permet de les réexécuter avec d'autres paramètres.

Structures des programmes développés
Soit CHRONO (i,n) l'ensemble des instructions permettant le chronométrage de l'instruction i exécuté n fois,
on a donc :
Programme :

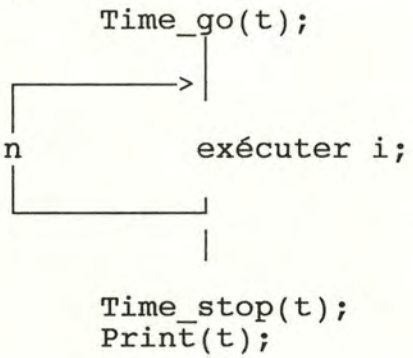


Les instructions i,j,...,k,l seront testées successivement 10.000, 100.000, 1.000.000, 10.000.000 fois. Seuls les résultats des tests sur 10.000.000 itérations sont reprises.

Structure de Chrono (i,n)
L'instruction à tester i est placée dans une boucle ou un ensemble de boucles imbriquées permettant d'arriver au nombre requis d'exécution (n) de l'instruction (i).



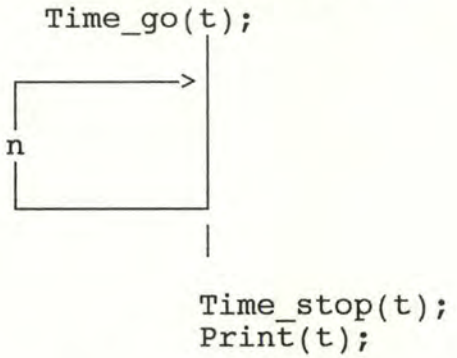
Sont placés autour de ce noyau d'instructions, les ensembles d'opérations mettant en place et déclenchant un processus de chronométrage (TIME_go (t) et TIME_stop (t)) et l'affichage des résultats (PRINT).



2.1.2 **Les programmes ADD et MUL**

Dans le rapport de mesures et performances, trois programmes ont été développés sur cette même structure :
1° pour tester les initialisations : INIT;
2° pour tester les additions : ADD;
3° pour tester les multiplications : MUL;
Les résultats des deux derniers programmes ont été repris dans ce mémoire.

Afin de mieux cerner les temps d'exécution des boucles elles-mêmes, des tests sur des boucles "vides" ont été effectués. Ces boucles sont de forme :



et ont été testées sans instructions internes. Tout ceci a été mis en oeuvre afin de mieux distinguer le temps d'exécution réel du bruit apporté par les instructions qui forment la boucle. Cette façon de travailler a amené à introduire un concept de test sur chacune des instructions "**en temps réel**", c'est-à-dire un temps d'exécution plus proche d'une certaine réalité, en d'autre termes égal à chrono (i,n) - chrono (ϕ,n).

Time_go(t) : - est une prise du temps courant grâce a *ftime()* (un appel système qui donne le temps écoulé en sec et ms (variables de type entier) depuis un temps).
- est une prise du temps processeur et système grâce à *times()* (un appel système qui renvoie ces deux temps en impulsions d'horloge).

Time_stop(t) : - reprend le temps courant (*ftime()*) et le temps "proc et syst" (*times()*), effectue la différence entre les temps pris dans time_go() et ceux juste enregistrés,
- modifie temps courant afin de l'obtenir en secondes et milli-secondes positives,
- transforme le temps exprimé en impulsions, en secondes et milli-secondes.

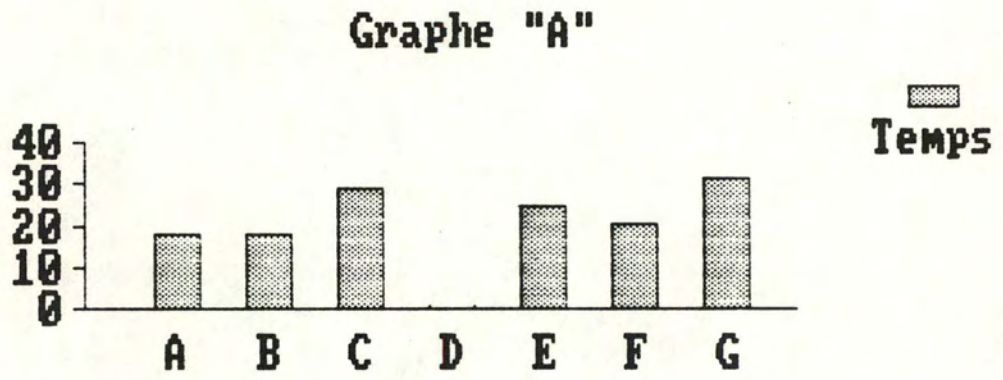
Print(t) : - affiche i et les temps totaux, processeurs et systèmes écoulés depuis le début de la boucle de répétition de l'instruction i jusqu'à sa fin.

Les tableaux avec les instructions individuelles :

Circé 08

Additions cas "A"

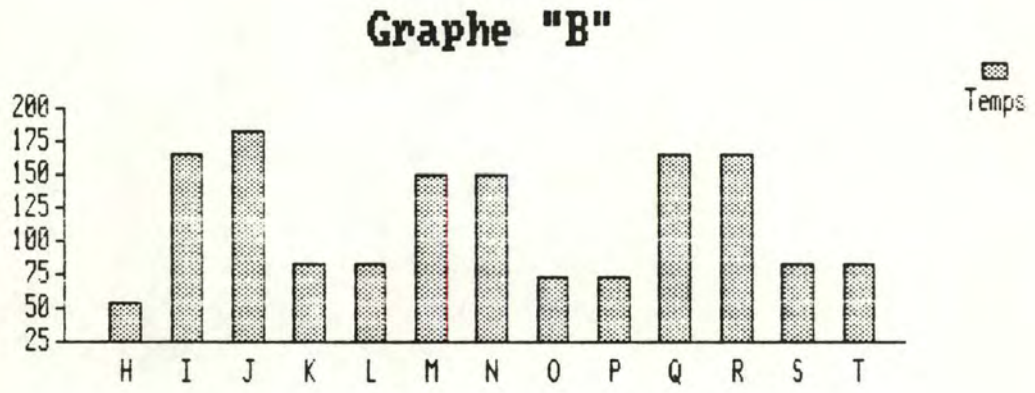
Id.	Instruction	Temps	Filtre
A	k++	17,92	A
B	k+=1	17,90	A
C	k=k+1	29,10	A
D	k+=0	,08	A
E	k=k+0	24,62	A
F	k+=13	20,16	A
G	k=k+13	31,36	A



Circé 08

Additions cas "B"

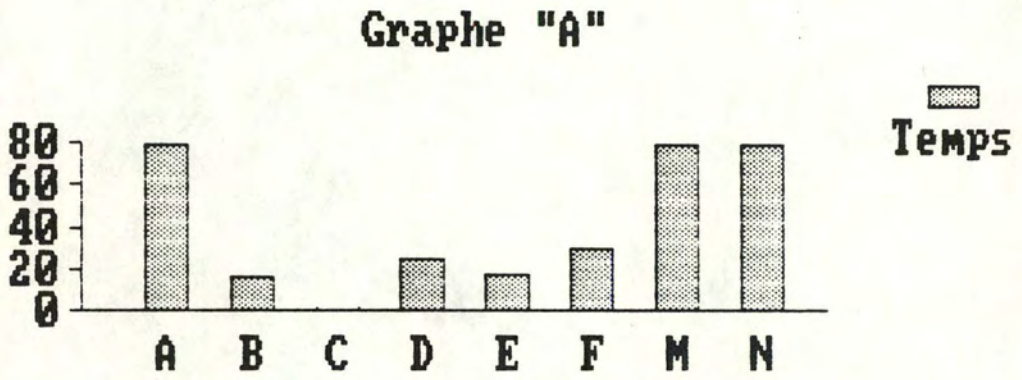
Id.	Instruction	Temps	Filtre
H	r++	53,72	B
I	r+=1	165,80	B
J	r+=1.	183,80	B
K	r=r+1	81,52	B
L	r=r+1.	81,54	B
M	r+=0	150,12	B
N	r+=0.	150,14	B
O	r=r+0	71,76	B
P	r=r+0.	71,70	B
Q	r+=13	165,80	B
R	r+=13.	165,80	B
S	r=r+13	81,62	B
T	r=r+13.	81,66	B



Circé 08

Multiplications cas "A"

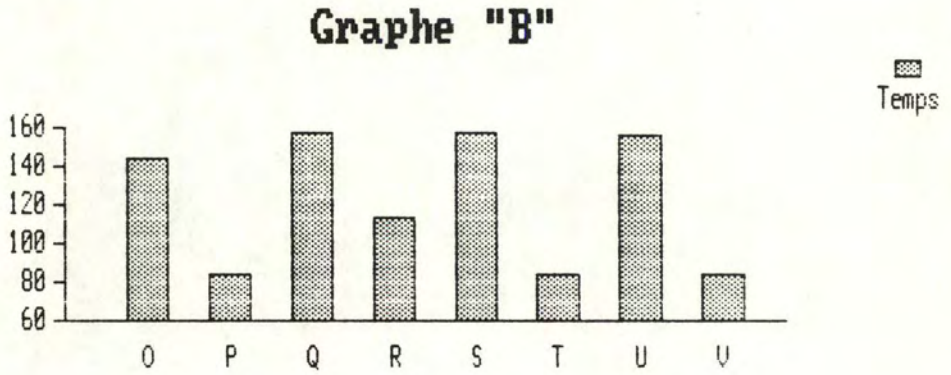
Id.	Instruction	Temps	Filtre
A	k*=0 [3]	78,38	A
B	k=k*0 [3]	15,68	A
C	k*=1 [3]	,02	A
D	k=k*1 [3]	24,68	A
E	k*=2 [3]	17,90	A
F	k=k*2 [3]	29,12	A
M	k*=13 [3]	78,44	A
N	k=k*13 [3]	78,42	A



Circé 08

Multiplications cas "B"

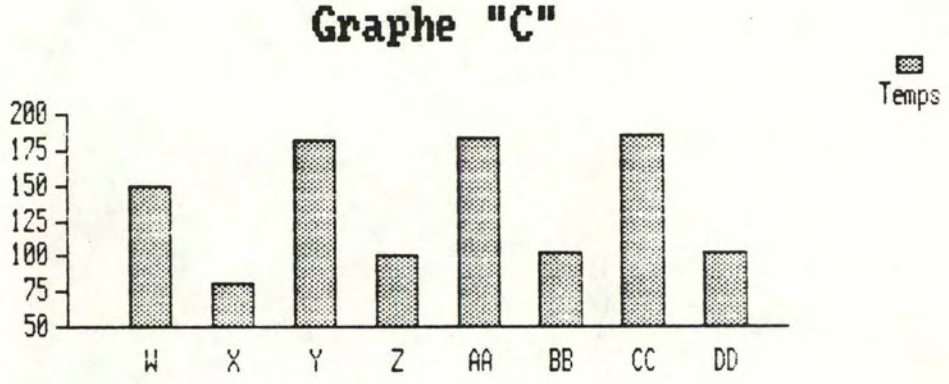
Id.	Instruction	Temps	Filtre
O	k*=0. [3]	143,58	B
P	k=k*0. [3]	84,04	B
Q	k*=1. [3]	156,84	B
R	k=k*1. [3]	113,16	B
S	k*=2. [3]	156,84	B
T	k=k*2. [3]	84,04	B
U	k*=13. [3]	155,72	B
V	k=k*13. [3]	84,20	B



Circé 08

Multiplications cas "C"

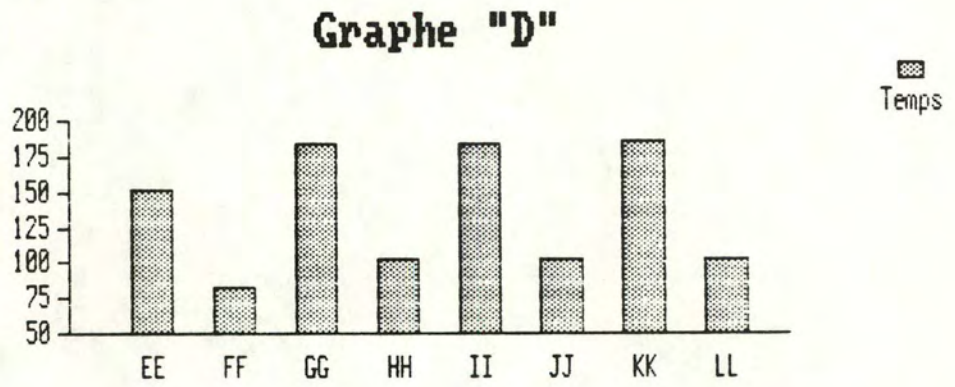
Id.	Instruction	Temps	Filtre
W	r*=0 [3]	150,12	C
X	r=r*0 [3]	80,66	C
Y	r*=1 [3]	182,64	C
Z	r=r*1 [3]	100,28	C
AA	r*=2 [2.1]	184,10	C
BB	r=r*2 [2.1]	101,72	C
CC	r*=05 [2.1]	185,28	C
DD	r=r*05 [2.1]	103,14	C



Circé 08

Multiplications cas "D"

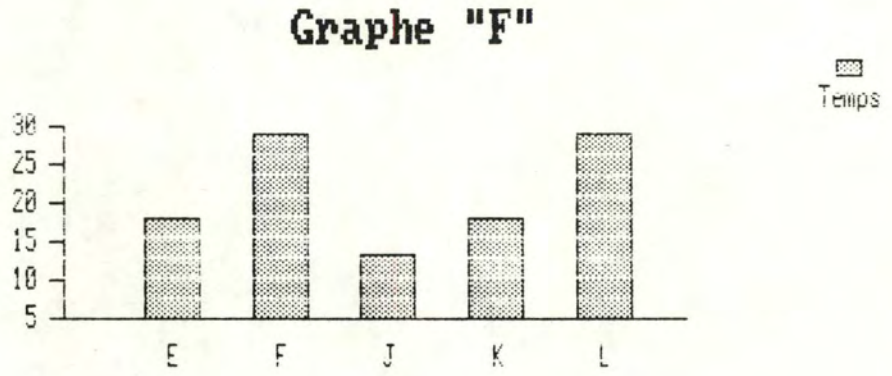
Id.	Instruction	Temps	Filtre
EE	r*=0. [2.1]	151,54	D
FF	r=r*0. [2.1]	82,10	D
GG	r*=1. [2.1]	184,08	D
HH	r=r*1. [2.1]	101,72	D
II	r*=2. [2.1]	184,12	D
JJ	r=r*2. [2.1]	101,82	D
KK	r*=05. [2.1]	185,28	D
LL	r=r*05. [2.1]	103,08	D



Circé 11

Multiplications cas "F"

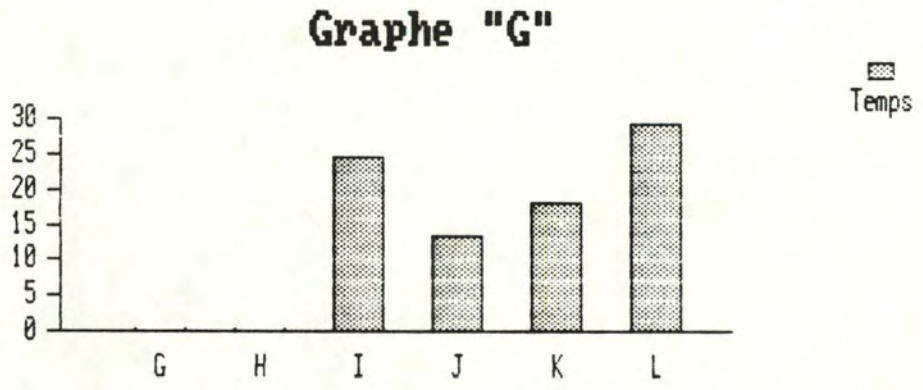
Id.	Instruction	Temps	Filtre
E	k*=2 [3]	17,92	F
F	k=k*2 [3]	29,16	F
J	k<<1 [3]	13,44	F
K	k<<=1 [3]	17,92	F
L	k=k<<1 [3]	29,12	F



Circé 08

Multiplications cas "G"

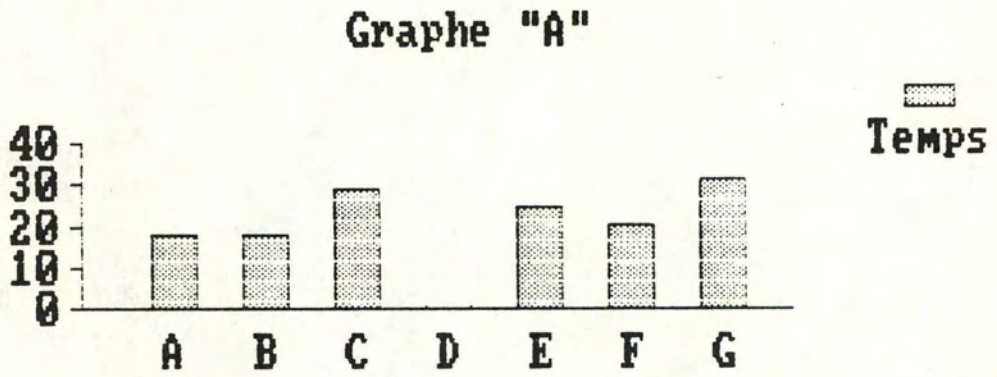
Id.	Instruction	Temps	Filtre
G	k<<0 [3]	0	G
H	k<<=0 [3]	,02	G
I	k=k<<0 [3]	24,66	G
J	k<<1 [3]	13,44	G
K	k<<=1 [3]	17,92	G
L	k=k<<1 [3]	29,12	G



Circé 11

Additions cas "A"

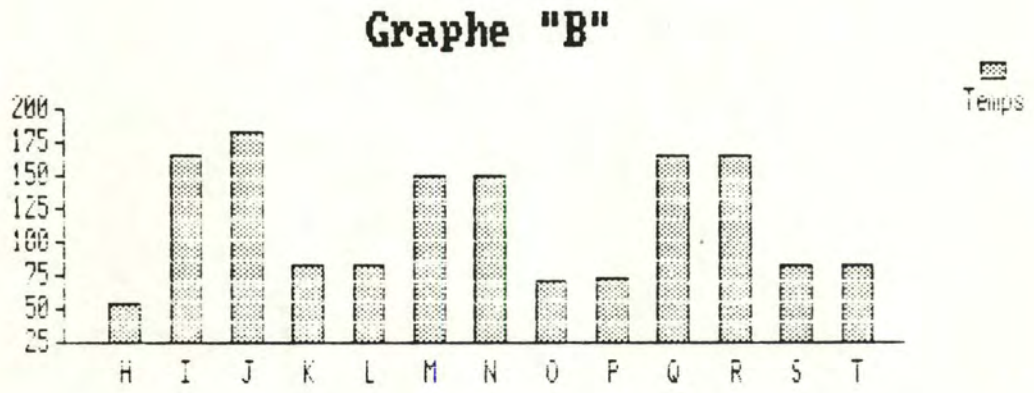
Id.	Instruction	Temps	Filtre
A	k++	17,92	A
B	k+=1	17,90	A
C	k=k+1	29,12	A
D	k+=0	,08	A
E	k=k+0	24,64	A
F	k+=13	20,16	A
G	k=k+13	31,36	A



Circé 11

Additions cas "B"

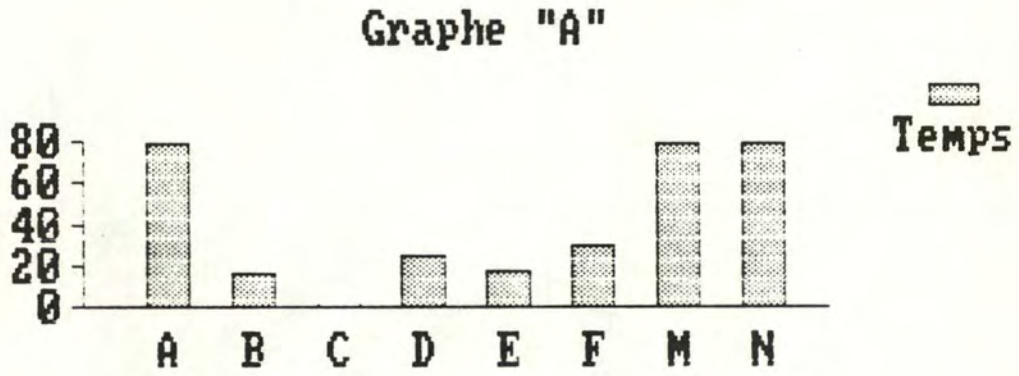
Id.	Instruction	Temps	Filtre
H	r++	53,76	B
I	r+=1	165,82	B
J	r+=1.	183,82	B
K	r=r+1	81,54	B
L	r=r+1.	81,54	B
M	r+=0	150,16	B
N	r+=0.	150,12	B
O	r=r+0	71,66	B
P	r=r+0.	71,80	B
Q	r+=13	165,98	B
R	r+=13.	165,82	B
S	r=r+13	81,68	B
T	r=r+13.	81,68	B



Circé 11

Multiplications cas "A"

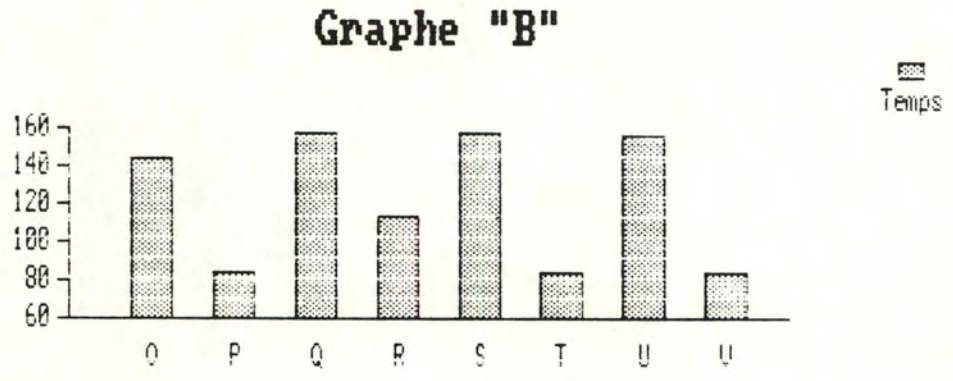
Id.	Instruction	Temps	Filtre
A	k*=0 [3]	78,42	A
B	k=k*0 [3]	15,70	A
C	k*=1 [3]	0	A
D	k=k*1 [3]	24,68	A
E	k*=2 [3]	17,92	A
F	k=k*2 [3]	29,16	A
M	k*=13 [3]	78,42	A
N	k=k*13 [3]	78,40	A



Circé 11

Multiplications cas "B"

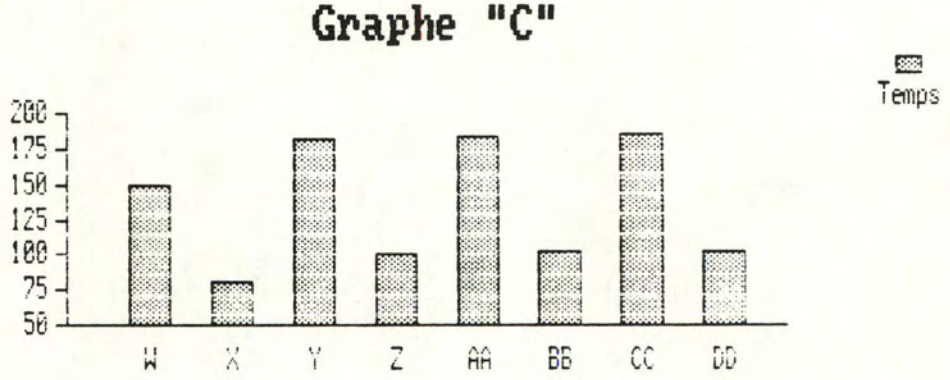
Id.	Instruction	Temps	Filtre
O	k*=0. [3]	143,58	B
P	k=k*0. [3]	84,02	B
Q	k*=1. [3]	156,82	B
R	k=k*1. [3]	113,12	B
S	k*=2. [3]	156,82	B
T	k=k*2. [3]	84,04	B
U	k*=13. [3]	155,72	B
V	k=k*13. [3]	84,20	B



Circé 11

Multiplications cas "C"

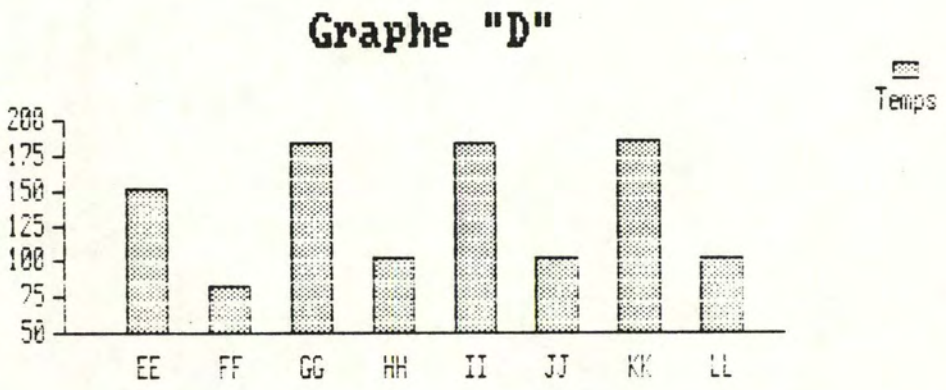
Id.	Instruction	Temps	Filtre
W	r*=0 [3]	150,12	C
X	r=r*0 [3]	80,68	C
Y	r*=1 [3]	182,64	C
Z	r=r*1 [3]	100,34	C
AA	r*=2 [2.1]	184,10	C
BB	r=r*2 [2.1]	101,72	C
CC	r*=05 [2.1]	185,28	C
DD	r=r*05 [2.1]	103,14	C



Circé 11

Multiplications cas "D"

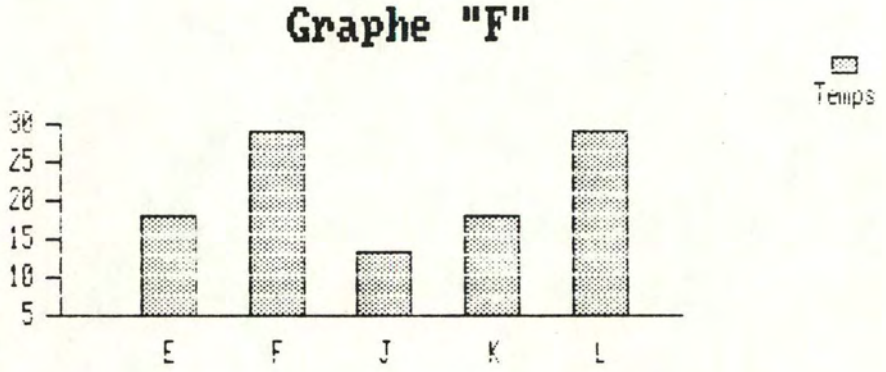
Id.	Instruction	Temps	Filtre
EE	r*=0. [2.1]	151,56	D
FF	r=r*0. [2.1]	82,10	D
GG	r*=1. [2.1]	184,12	D
HH	r=r*1. [2.1]	101,74	D
II	r*=2. [2.1]	184,12	D
JJ	r=r*2. [2.1]	101,84	D
KK	r*=05. [2.1]	185,30	D
LL	r=r*05. [2.1]	103,04	D



Circé 11

Multiplications cas "F"

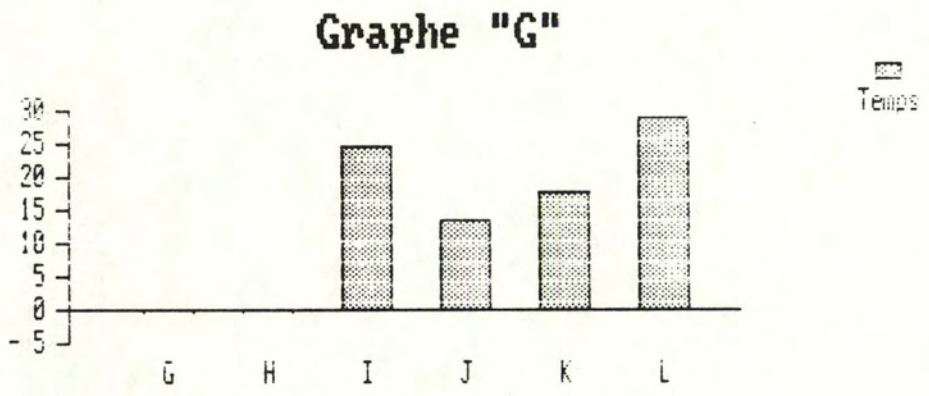
Id.	Instruction	Temps	Filtre
E	k*=2 [3]	17,92	F
F	k=k*2 [3]	29,16	F
J	k<<1 [3]	13,44	F
K	k<<=1 [3]	17,92	F
L	k=k<<1 [3]	29,12	F



Circé 11

Multiplications cas "G"

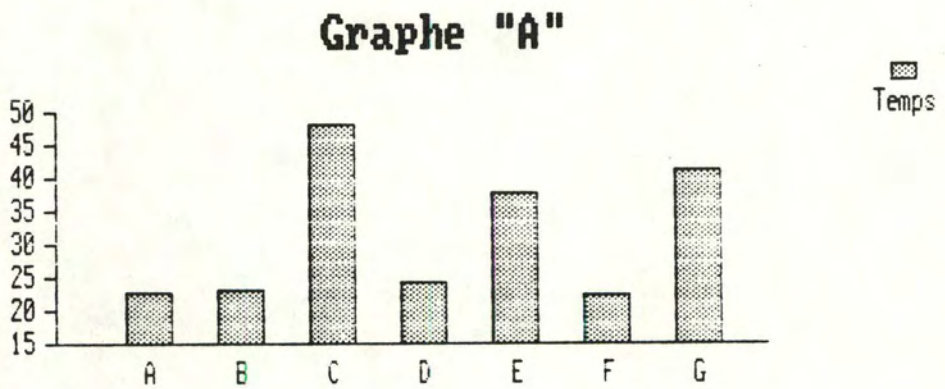
Id.	Instruction	Temps	Filtre
G	k<<0 [3]	-,02	G
H	k<<=0 [3]	-,02	G
I	k=k<<0 [3]	24,66	G
J	k<<1 [3]	13,44	G
K	k<<=1 [3]	17,92	G
L	k=k<<1 [3]	29,12	G



Alma 11

Additions cas "A"

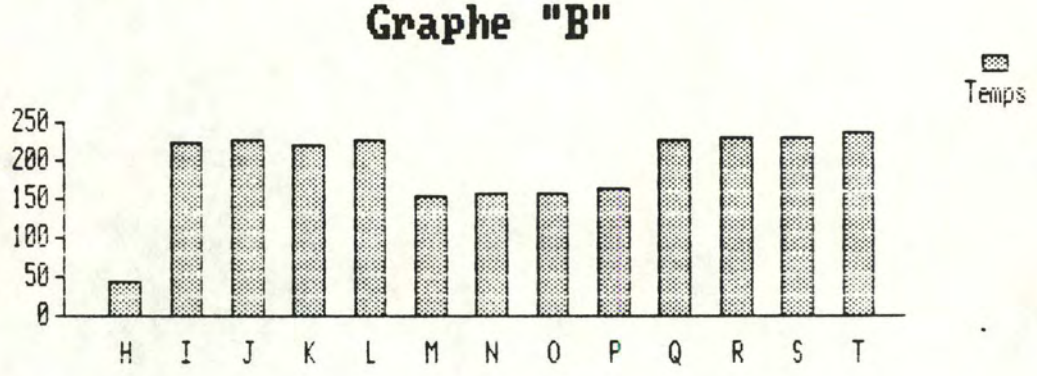
Id.	Instruction	Temps	Filtre
A	k++	22,40	A
B	k+=1	22,80	A
C	k=k+1	48,43	A
D	k+=0	24,12	A
E	k=k+0	37,87	A
F	k+=13	22,30	A
G	k=k+13	41,38	A



Alma 11

Additions cas "B"

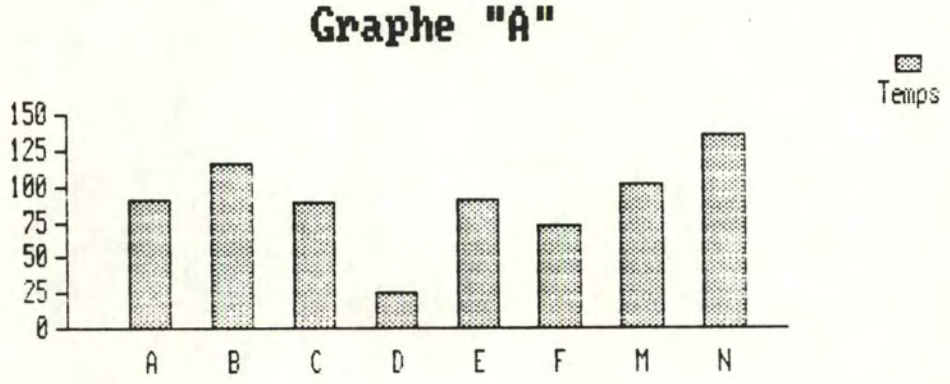
Id.	Instruction	Temps	Filtre
H	r++	43,12	B
I	r+=1	222,13	B
J	r+=1.	226,62	B
K	r=r+1	220,60	B
L	r=r+1.	225,35	B
M	r+=0	154,08	B
N	r+=0.	157,63	B
O	r=r+0	157,87	B
P	r=r+0.	162,65	B
Q	r+=13	227,75	B
R	r+=13.	231,12	B
S	r=r+13	231,48	B
T	r=r+13.	236,10	B



Alma 11

Multiplications cas "A"

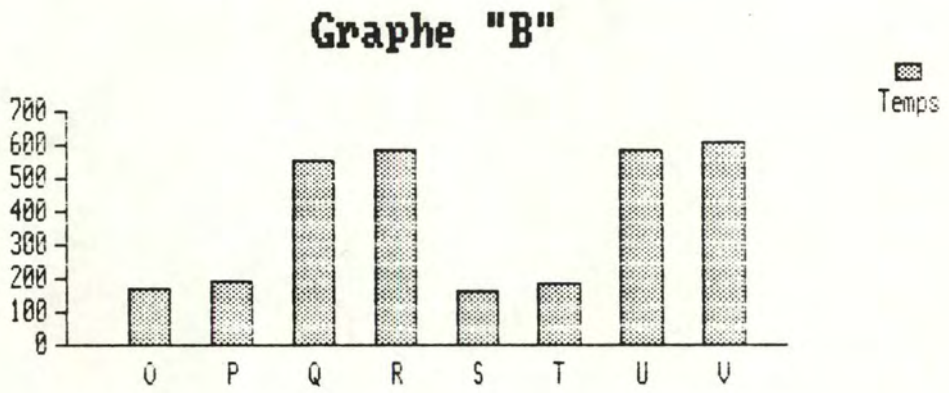
Id.	Instruction	Temps	Filtre
A	k*=0 [3]	89,95	A
B	k=k*0 [3]	115,72	A
C	k*=1 [3]	88,27	A
D	k=k*1 [3]	25,92	A
E	k*=2 [3]	89,95	A
F	k=k*2 [3]	72,60	A
M	k*=13 [3]	101,07	A
N	k=k*13 [3]	135,28	A



Alma 11

Multiplications cas "B"

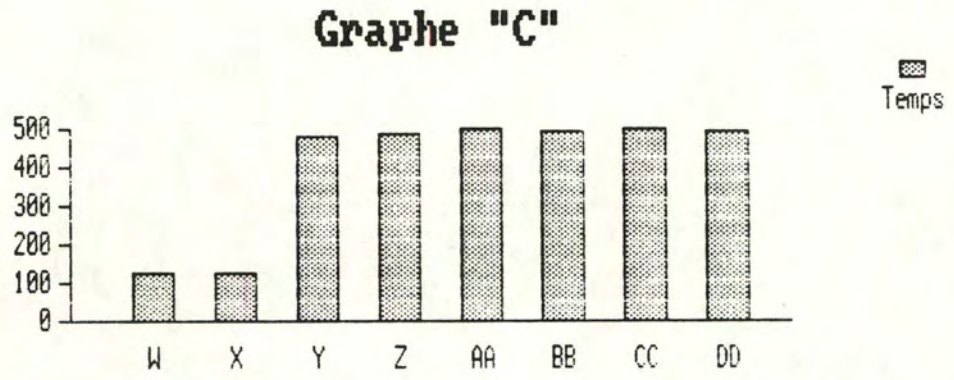
Id.	Instruction	Temps	Filtre
O	k*=0. [3]	166,05	B
P	k=k*0. [3]	187,27	B
Q	k*=1. [3]	557,98	B
R	k=k*1. [3]	583,70	B
S	k*=2. [3]	155,82	B
T	k=k*2. [3]	183,27	B
U	k*=13. [3]	587,75	B
V	k=k*13. [3]	610,08	B



Alma 11

Multiplications cas "C"

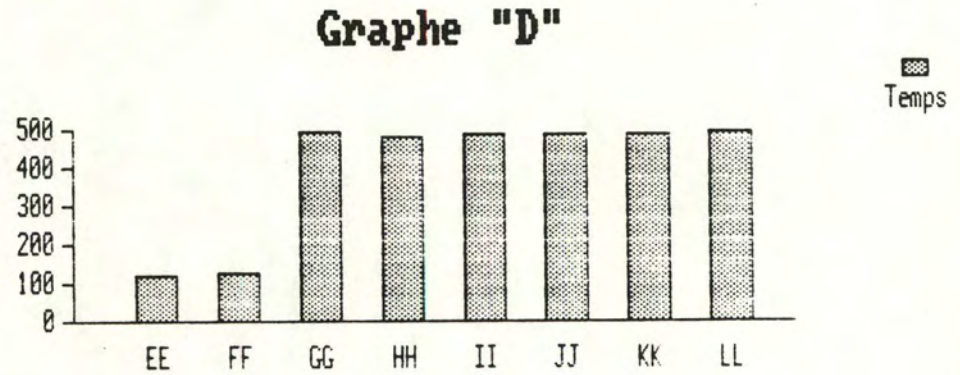
Id.	Instruction	Temps	Filtre
W	r*=0 [3]	128,05	C
X	r=r*0 [3]	125,40	C
Y	r*=1 [3]	479,15	C
Z	r=r*1 [3]	483,72	C
AA	r*=2 [2.1]	499,27	C
BB	r=r*2 [2.1]	494,65	C
CC	r*=05 [2.1]	498,70	C
DD	r=r*05 [2.1]	495,93	C



Alma 11

Multiplications cas "D"

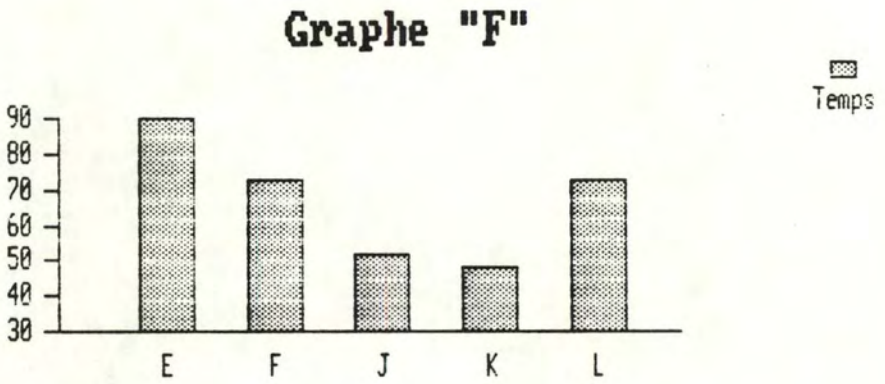
Id.	Instruction	Temps	Filtre
EE	r*=0. [2.1]	123,32	D
FF	r=r*0. [2.1]	124,28	D
GG	r*=1. [2.1]	492,23	D
HH	r=r*1. [2.1]	483,18	D
II	r*=2. [2.1]	488,32	D
JJ	r=r*2. [2.1]	488,17	D
KK	r*=05. [2.1]	487,25	D
LL	r=r*05. [2.1]	491,07	D



Alma 11

Multiplications cas "F"

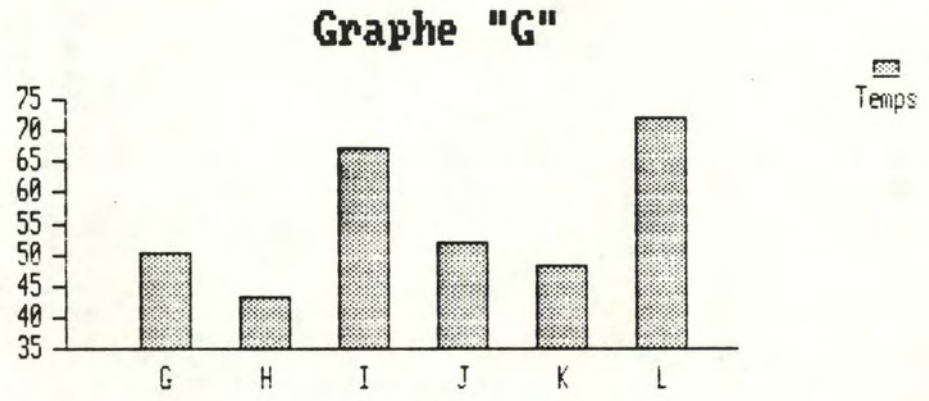
Id.	Instruction	Temps	Filtre
E	k*=2 [3]	89,95	F
F	k=k*2 [3]	72,60	F
J	k<<1 [3]	51,88	F
K	k<<=1 [3]	48,25	F
L	k=k<<1 [3]	72,45	F



Alma 11

Multiplications cas "G"

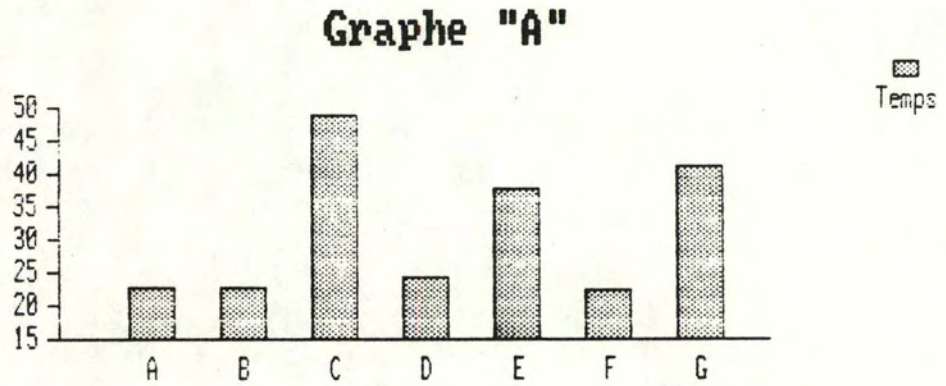
Id.	Instruction	Temps	Filtre
G	k<<0 [3]	50,13	G
H	k<<=0 [3]	43,05	G
I	k=k<<0 [3]	67,35	G
J	k<<1 [3]	51,88	G
K	k<<=1 [3]	48,25	G
L	k=k<<1 [3]	72,45	G



Alma 12

Additions cas "A"

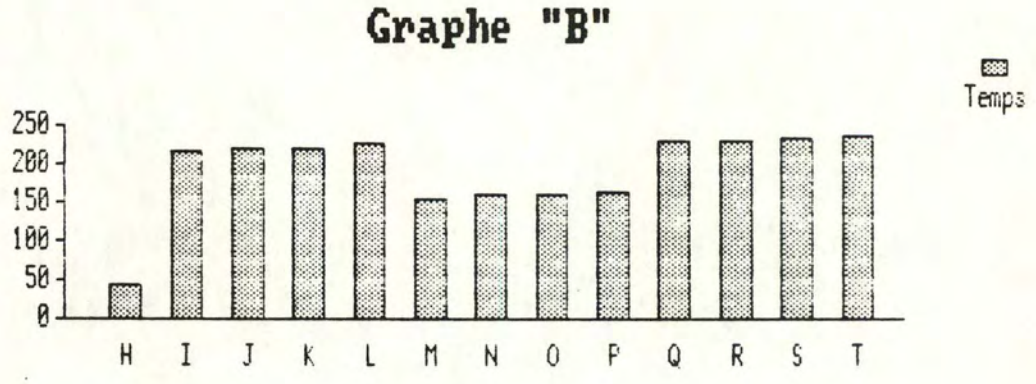
Id.	Instruction	Temps	Filtre
A	k++	22,63	A
B	k+=1	22,73	A
C	k=k+1	49,17	A
D	k+=0	24,13	A
E	k=k+0	37,93	A
F	k+=13	22,32	A
G	k=k+13	41,42	A



Alma 12

Additions cas "B"

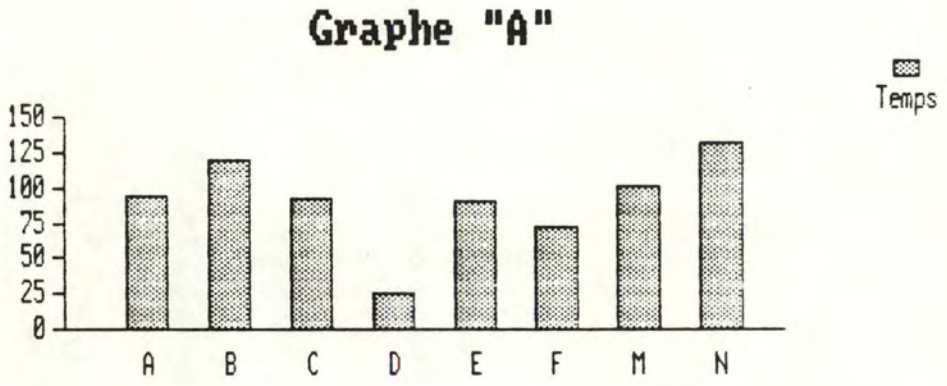
Id.	Instruction	Temps	Filtre
H	r++	43,42	B
I	r+=1	217,12	B
J	r+=1.	219,90	B
K	r=r+1	219,98	B
L	r=r+1.	225,27	B
M	r+=0	153,78	B
N	r+=0.	159,75	B
O	r=r+0	158,67	B
P	r=r+0.	162,85	B
Q	r+=13	229,23	B
R	r+=13.	231,40	B
S	r=r+13	233,15	B
T	r=r+13.	236,47	B



Alma 12

Multiplications cas "A"

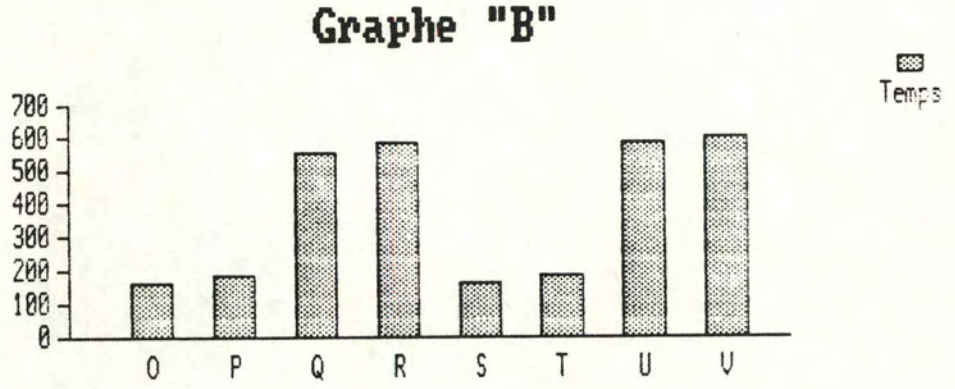
Id.	Instruction	Temps	Filtre
A	k*=0 [3]	93,62	A
B	k=k*0 [3]	119,80	A
C	k*=1 [3]	91,57	A
D	k=k*1 [3]	25,70	A
E	k*=2 [3]	89,73	A
F	k=k*2 [3]	72,40	A
M	k*=13 [3]	100,85	A
N	k=k*13 [3]	132,28	A



Alma 12

Multiplications cas "B"

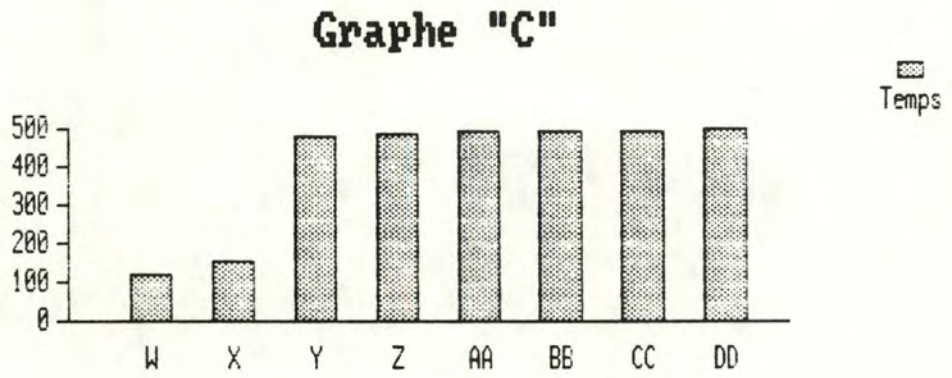
Id.	Instruction	Temps	Filtre
O	k*=0. [3]	160,47	B
P	k=k*0. [3]	181,43	B
Q	k*=1. [3]	557,37	B
R	k=k*1. [3]	583,90	B
S	k*=2. [3]	155,70	B
T	k=k*2. [3]	183,25	B
U	k*=13. [3]	586,68	B
V	k=k*13. [3]	603,77	B



Alma 12

Multiplications cas "C"

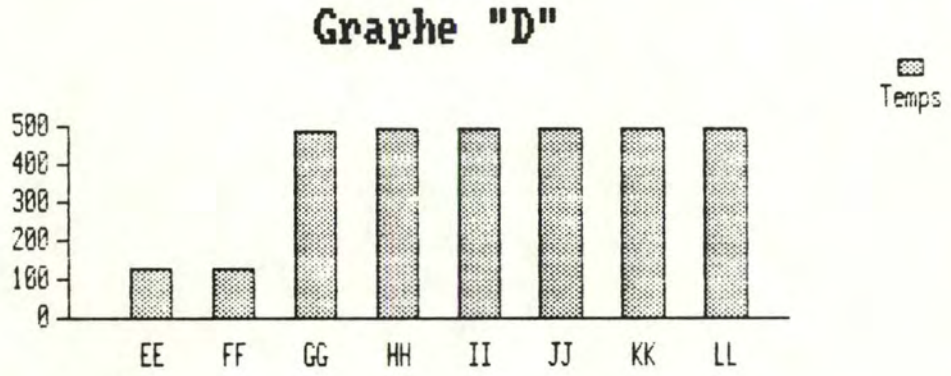
Id.	Instruction	Temps	Filtre
W	r*=0 [3]	123	C
X	r=r*0 [3]	152,23	C
Y	r*=1 [3]	480,75	C
Z	r=r*1 [3]	486,38	C
AA	r*=2 [2.1]	492,85	C
BB	r=r*2 [2.1]	494,13	C
CC	r*=05 [2.1]	492,52	C
DD	r=r*05 [2.1]	497,63	C



Alma 12

Multiplications cas "D"

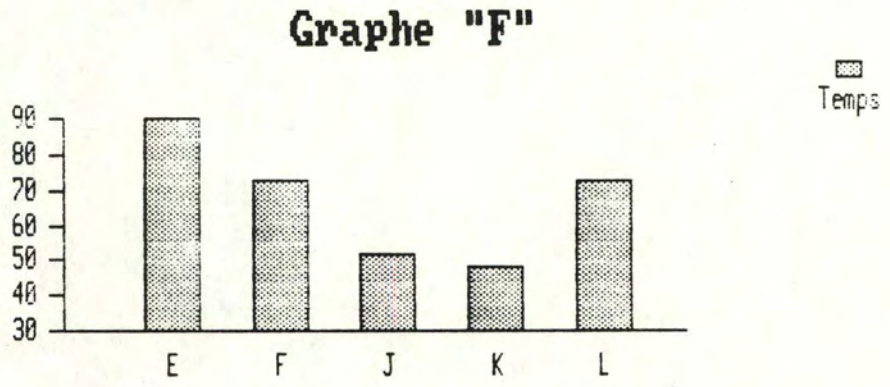
Id.	Instruction	Temps	Filtre
EE	r*=0. [2.1]	124,18	D
FF	r=r*0. [2.1]	124,08	D
GG	r*=1. [2.1]	487,02	D
HH	r=r*1. [2.1]	490,37	D
II	r*=2. [2.1]	491,40	D
JJ	r=r*2. [2.1]	492,35	D
KK	r*=05. [2.1]	490,22	D
LL	r=r*05. [2.1]	495,52	D



Alma 12

Multiplications cas "F"

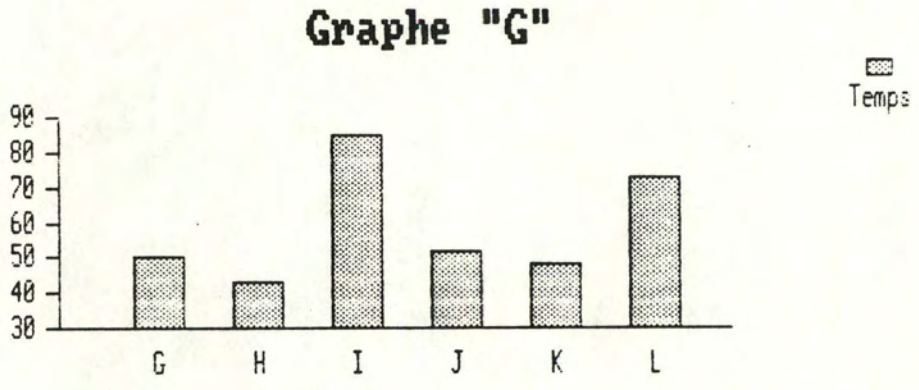
Id.	Instruction	Temps	Filtre
E	k*=2 [3]	89,73	F
F	k=k*2 [3]	72,40	F
J	k<<1 [3]	51,65	F
K	k<<=1 [3]	48,10	F
L	k=k<<1 [3]	72,40	F



Alma 12

Multiplications cas "G"

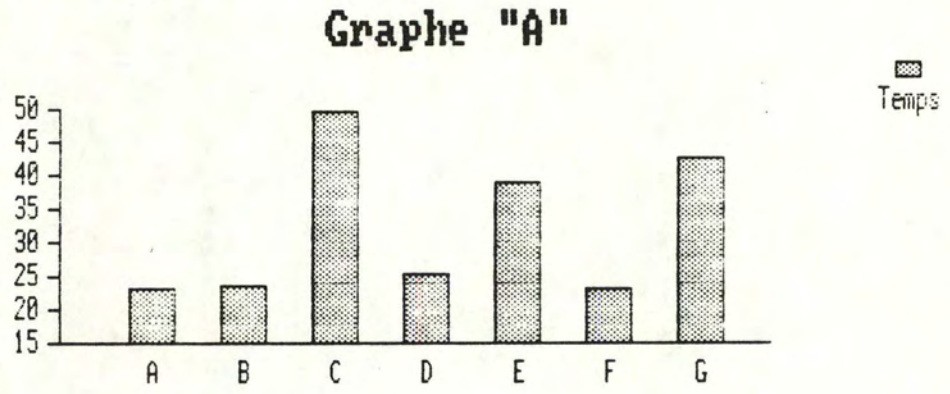
Id.	Instruction	Temps	Filtre
G	k<<0 [3]	49,93	G
H	k<<=0 [3]	42,77	G
I	k=k<<0 [3]	85,18	G
J	k<<1 [3]	51,65	G
K	k<<=1 [3]	48,10	G
L	k=k<<1 [3]	72,40	G



Junon 11

Additions cas "A"

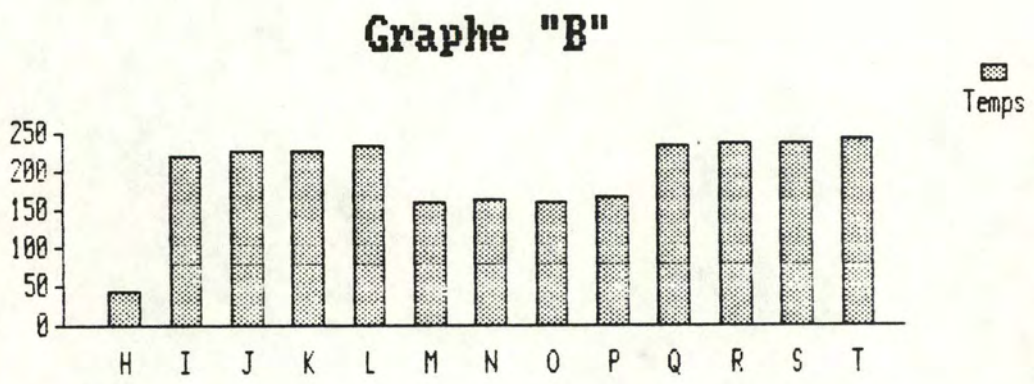
Id.	Instruction	Temps	Filtre
A	k++	23,08	A
B	k+=1	23,42	A
C	k=k+1	49,78	A
D	k+=0	25,12	A
E	k=k+0	38,88	A
F	k+=13	23,13	A
G	k=k+13	42,75	A



Junon 11

Additions cas "B"

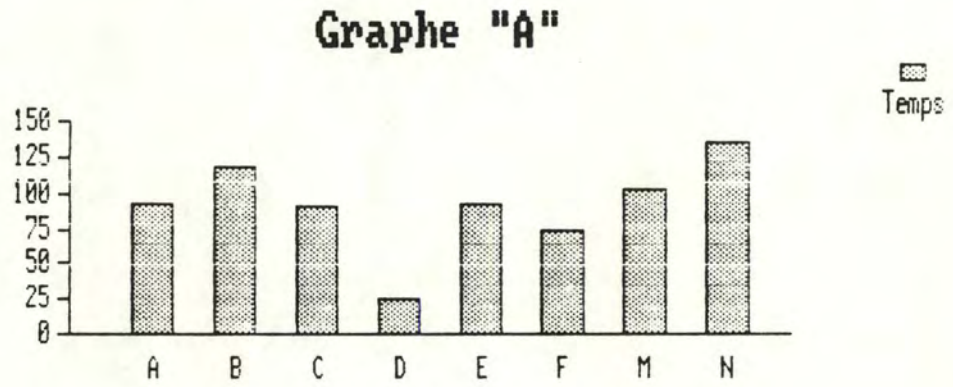
Id.	Instruction	Temps	Filtre
H	r++	44,43	B
I	r+=1	221,33	B
J	r+=1.	226	B
K	r=r+1	227,77	B
L	r=r+1.	231,65	B
M	r+=0	159,70	B
N	r+=0.	164,03	B
O	r=r+0	161,60	B
P	r=r+0.	167,90	B
Q	r+=13	232,25	B
R	r+=13.	236,73	B
S	r=r+13	237,15	B
T	r=r+13.	242,67	B



Junon 11

Multiplications cas "A"

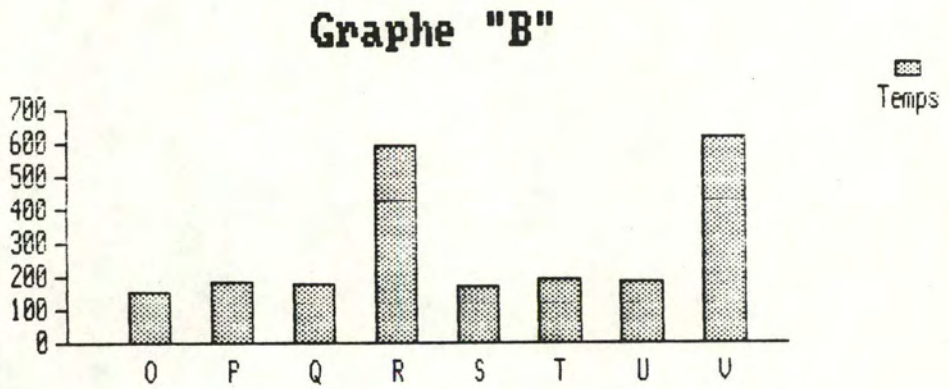
Id.	Instruction	Temps	Filtre
A	k*=0 [3]	91,40	A
B	k=k*0 [3]	117,70	A
C	k*=1 [3]	89,83	A
D	k=k*1 [3]	26,15	A
E	k*=2 [3]	91,28	A
F	k=k*2 [3]	73,87	A
M	k*=13 [3]	103,05	A
N	k=k*13 [3]	134,68	A



Junon 11

Multiplications cas "B"

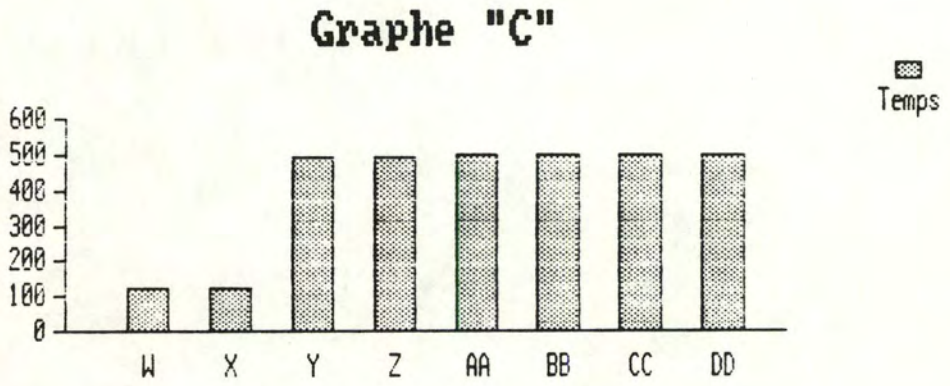
Id.	Instruction	Temps	Filtre
O	k*=0. [3]	151,67	B
P	k=k*0. [3]	185	B
Q	k*=1. [3]	174,43	B
R	k=k*1. [3]	595,33	B
S	k*=2. [3]	170,78	B
T	k=k*2. [3]	187,10	B
U	k*=13. [3]	180,60	B
V	k=k*13. [3]	615,02	B



Junon 11

Multiplications cas "C"

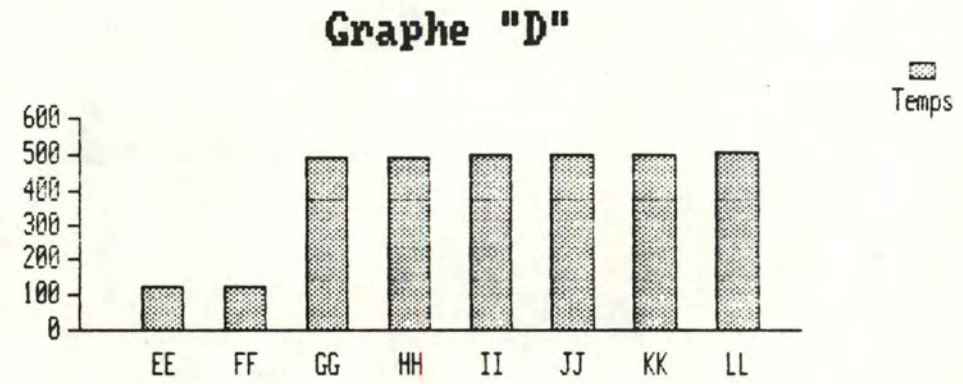
Id.	Instruction	Temps	Filtre
W	r*=0 [3]	124,75	C
X	r=r*0 [3]	124,97	C
Y	r*=1 [3]	489,50	C
Z	r=r*1 [3]	493,28	C
AA	r*=2 [2.1]	499,63	C
BB	r=r*2 [2.1]	498,92	C
CC	r*=05 [2.1]	497,38	C
DD	r=r*05 [2.1]	501,65	C



Junon 11

Multiplications cas "D"

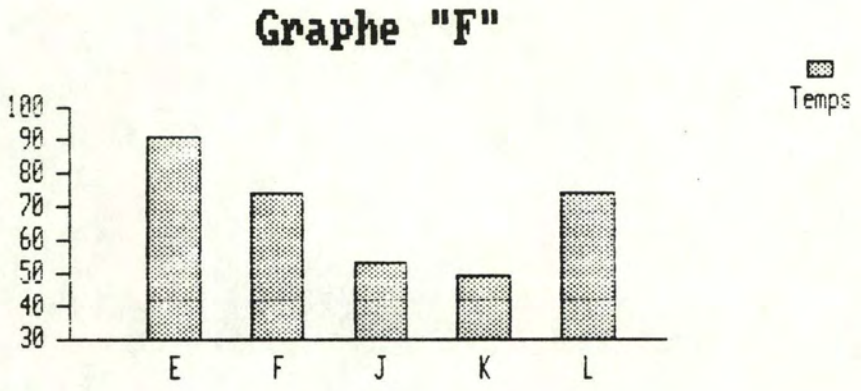
Id.	Instruction	Temps	Filtre
EE	r*=0. [2.1]	126,13	D
FF	r=r*0. [2.1]	125,95	D
GG	r*=1. [2.1]	490,37	D
HH	r=r*1. [2.1]	493,88	D
II	r*=2. [2.1]	499,03	D
JJ	r=r*2. [2.1]	500,98	D
KK	r*=05. [2.1]	498,02	D
LL	r=r*05. [2.1]	503,18	D



Junon 11

Multiplications cas "F"

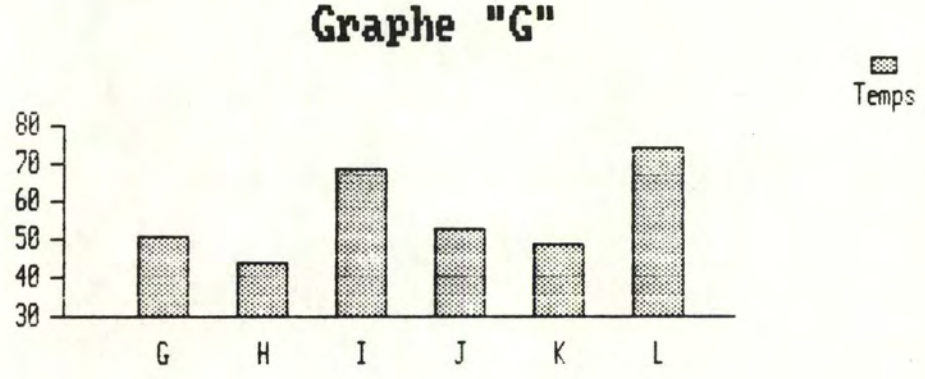
Id.	Instruction	Temps	Filtre
E	k*=2 [3]	91,28	F
F	k=k*2 [3]	73,87	F
J	k<<1 [3]	52,87	F
K	k<<=1 [3]	48,97	F
L	k=k<<1 [3]	73,73	F



Junon 11

Multiplications cas "G"

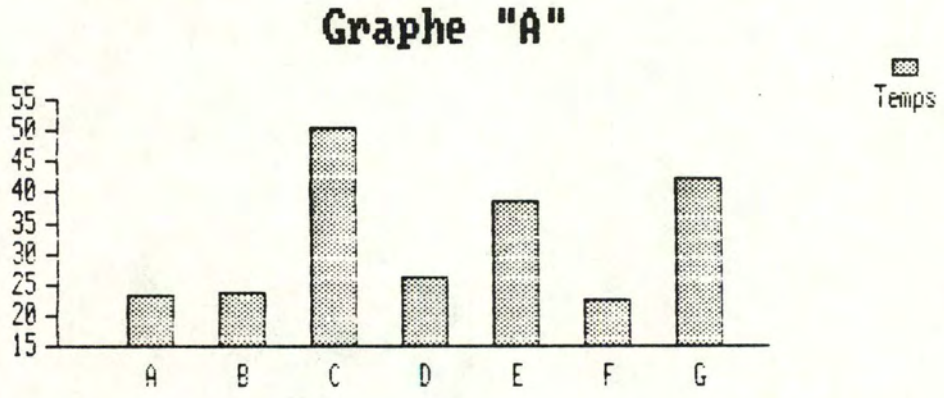
Id.	Instruction	Temps	Filtre
G	k<<0 [3]	50,92	G
H	k<<=0 [3]	43,68	G
I	k=k<<0 [3]	68,38	G
J	k<<1 [3]	52,87	G
K	k<<=1 [3]	48,97	G
L	k=k<<1 [3]	73,73	G



Junon 12

Additions cas "A"

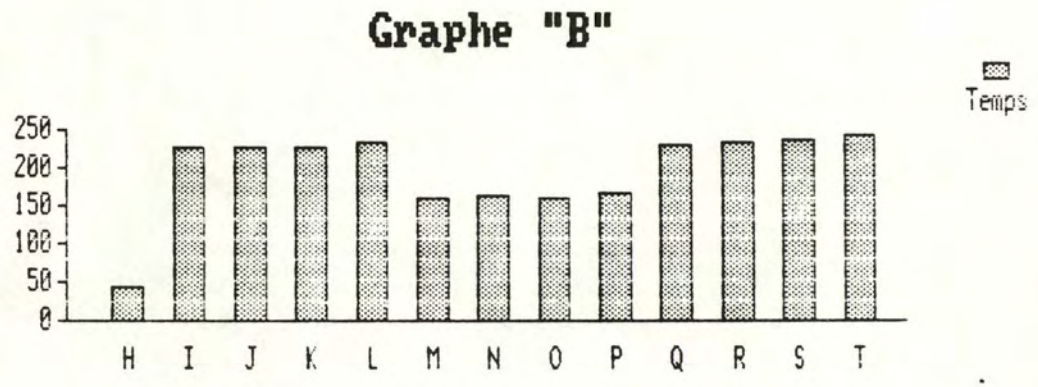
Id.	Instruction	Temps	Filtre
A	k++	23,30	A
B	k+=1	23,58	A
C	k=k+1	50,42	A
D	k+=0	25,90	A
E	k=k+0	38,43	A
F	k+=13	22,42	A
G	k=k+13	42,25	A



Junon 12

Additions cas "B"

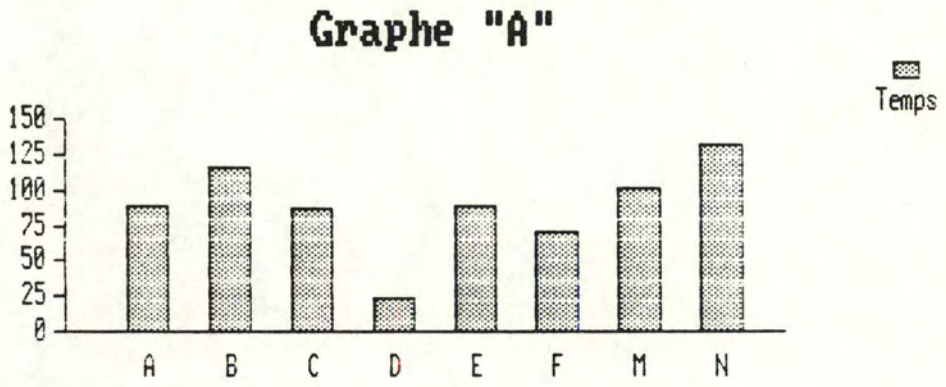
Id.	Instruction	Temps	Filtre
H	r+=	44,05	B
I	r+=1	225,12	B
J	r+=1.	227,55	B
K	r=r+1	227,13	B
L	r=r+1.	234,93	B
M	r+=0	159,12	B
N	r+=0.	162,35	B
O	r=r+0	161,17	B
P	r=r+0.	167,35	B
Q	r+=13	230,80	B
R	r+=13.	234,02	B
S	r=r+13	238,13	B
T	r=r+13.	241,70	B



Junon 12

Multiplications cas "A"

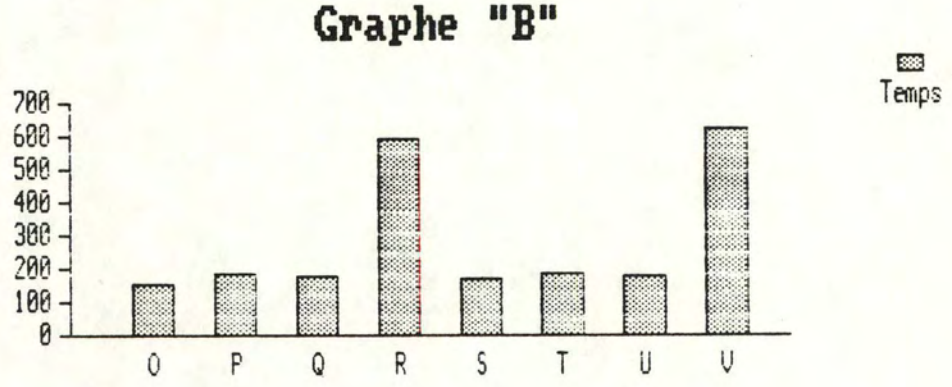
Id.	Instruction	Temps	Filtre
A	k*=0 [3]	88,73	A
B	k=k*0 [3]	115,02	A
C	k*=1 [3]	87,20	A
D	k=k*1 [3]	23,65	A
E	k*=2 [3]	88,53	A
F	k=k*2 [3]	71,08	A
M	k*=13 [3]	100,47	A
N	k=k*13 [3]	132	A



Junon 12

Multiplications cas "B"

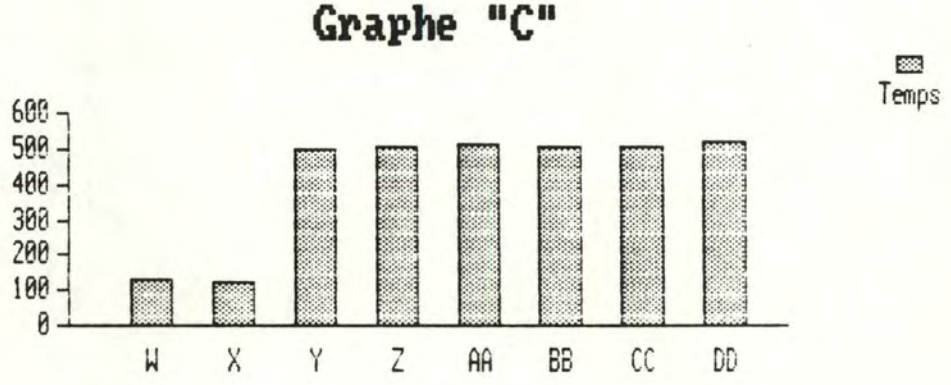
Id.	Instruction	Temps	Filtre
O	k*=0. [3]	149,07	B
P	k=k*0. [3]	182,52	B
Q	k*=1. [3]	171,77	B
R	k=k*1. [3]	591,58	B
S	k*=2. [3]	167,95	B
T	k=k*2. [3]	183,97	B
U	k*=13. [3]	177,77	B
V	k=k*13. [3]	625,52	B



Junon 12

Multiplications cas "C"

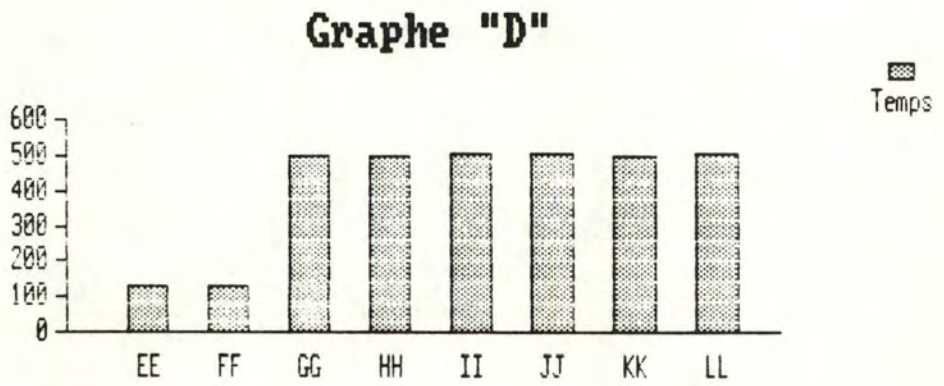
Id.	Instruction	Temps	Filtre
W	r*=0 [3]	129,42	C
X	r=r*0 [3]	125,72	C
Y	r*=1 [3]	496,40	C
Z	r=r*1 [3]	508,72	C
AA	r*=2 [2.1]	512,13	C
BB	r=r*2 [2.1]	506,78	C
CC	r*=05 [2.1]	506,25	C
DD	r=r*05 [2.1]	519,77	C



Junon 12

Multiplications cas "D"

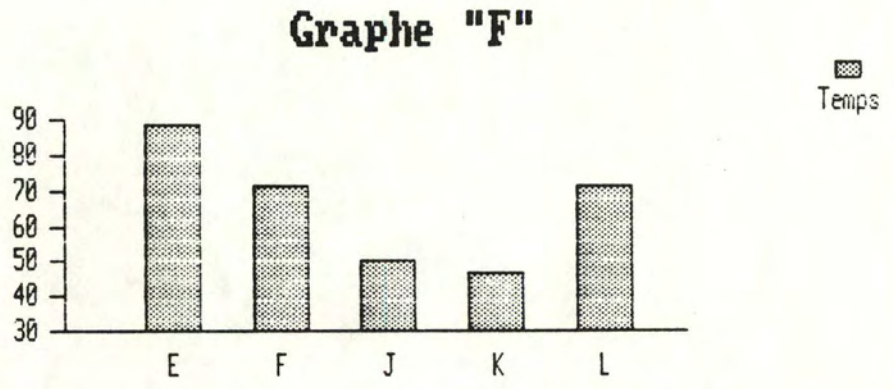
Id.	Instruction	Temps	Filtre
EE	r*=0. [2.1]	129,53	D
FF	r=r*0. [2.1]	129,12	D
GG	r*=1. [2.1]	498,22	D
HH	r=r*1. [2.1]	500,70	D
II	r*=2. [2.1]	503,18	D
JJ	r=r*2. [2.1]	503,35	D
KK	r*=05. [2.1]	501,83	D
LL	r=r*05. [2.1]	508,03	D



Junon 12

Multiplications cas "F"

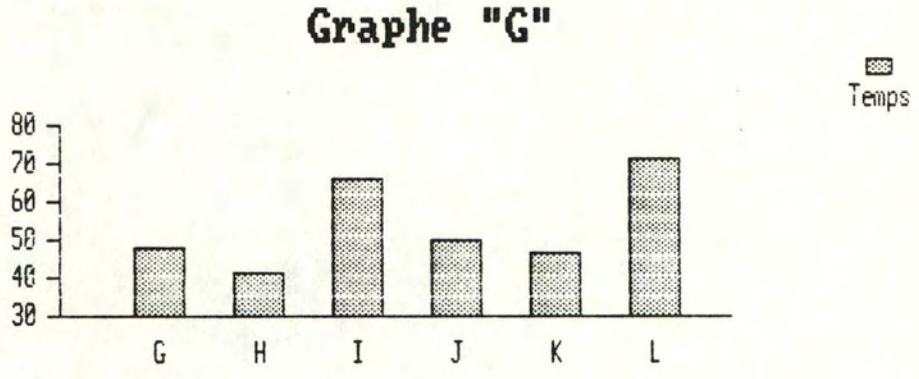
Id.	Instruction	Temps	Filtre
E	k*=2 [3]	88,53	F
F	k=k*2 [3]	71,08	F
J	k<<1 [3]	50,07	F
K	k<<=1 [3]	46,63	F
L	k=k<<1 [3]	71,37	F



Junon 12

Multiplications cas "G"

Id.	Instruction	Temps	Filtre
G	k<<0 [3]	48,32	G
H	k<<=0 [3]	41,22	G
I	k=k<<0 [3]	65,75	G
J	k<<1 [3]	50,07	G
K	k<<=1 [3]	46,63	G
L	k=k<<1 [3]	71,37	G



2.3 Les programmes sources

2.3.1 Remarque préalable

Les programmes ci-joints forment un résumé des programmes réels, c'est-à-dire que chaque occurrence de "**Instruction testée**" doit être remplacée par l'instruction C correspondante.

2.3.2 Texte du programme

```
#include <stdio.h>
#include <sys/types.h>
#define NBE_MIL          10

main()
{
#include <sys/timeb.h>
#include <sys/times.h>

char      *ctime(), malloc();
time_t    time(), *tloc1, *tloc2;

struct    timeb *buffer;
struct    tms   *tampon;

float     nfois, res, res_u, res_s, r;
long      tms_u1, tms_u2, tms_s1, tms_s2;
int       sec1, sec2, ecart_sec, ecart_ms, i, j, k, l, m;
int       boucle, ms1, ms2;

tloc1 = ((time_t *) malloc(sizeof(time_t)));
tloc2 = ((time_t *) malloc(sizeof(time_t)));
buffer = ((struct timeb *) malloc(sizeof(struct timeb)));
tampon = ((struct timeb *) malloc(sizeof(struct timeb)));
if ( tloc1 == NULL) exit(1);
if ( tloc2 == NULL) exit(1);
if ( buffer == NULL) exit(1);
if ( tampon == NULL) exit(1);

nfois = NBE_MIL * 10.;
for(boucle = 1; boucle <= 4; boucle++)
{
nfois = nfois * 10.;
l = (nfois / 1000);
*tloc2 = time(tloc1);

printf("date :                %s\n", ctime(tloc2));
printf("nombre d'itérations : %10.0f\n\n\n", nfois);
printf("   instruction testée      pour %10.0f it.      \n",
                                        nfois);
printf("   instruction testée      temps total          temps
proc      temps sys\n");
printf("---------------------¦-----------------¦-------------
```

```
-------¦----------\n");

/*---------------------- les programmes --------------------*/

ftime(buffer);
sec1 = (*buffer).time;
ms1 = (*buffer).millitm;

times(tampon);
tms_u1 = (*tampon).tms_utime;
tms_u2 = (*tampon).tms_stime;

     for(i = 1; i <= 1; i++) {

          for(j = 1; j <= 20; i++) {

               for(j = 1; j <= 50; j++) {

               Instruction testée
               }
          }
     }

ftime(buffer);
sec2 = (*buffer).time;
m2 = (*buffer).millitm;

times(tampon);
tms_u1 = (*tampon).tms_utime;
tms_u2 = (*tampon).tms_stime;

ecart_sec = sec2 - sec1;
ecart_ms = ms2 - ms1;
if ( ecart_ms < 0) {
     ecart_sec -= 1;
     ecart_ms += 1000;
     }

res = (ecart_sec * 1000. + ecart_ms) / 1000.;
res_u = (tms_u2 - tms_u1) / 50.;
res_u = (tms_s2 - tms_s1) / 50.;

printf(" printf("   Instruction testée ¦          %7.3f    ¦
     %7.3f      ¦     %7.3f\n", res, res_u, res_s);

} /* fin de boucle principale */

} /* fin de programme */
```

3 Les tests sur les performances des programmes complets

3.1 Méthode de prise de mesures

Les valeurs contenues dans les tableaux "Tableau récapitulatif des tailles" et "Tableau récapitulatif des moyennes" ont été obtenues de la manière suivante : toutes les valeurs sont en secondes; on a pour chaque valeur, utilisé cinq mesures et calculé la moyenne.

Les valeurs dans la colonne "Unix Exec" ont été obtenues sur Unix, avec la commande "time <nom de programme>". Cette commande renvoit trois temps en secondes, c'est-à-dire, le real time ou le temps réel à partir de l'envoi de la commande jusqu'à la réponse (le retour du prompt), le user time ou le temps utilisé par le processus pour exécuter des instructions non-privilégiées le sys time ou le temps système ou le temps utilisé par le processus pour exécuter des instructions privilégiées (des instructions système). Le temps CPU pour l'exécution du programme a été calculé comme la somme des user time et sys time. Le temps réel n'a pas été retenu, étant donné qu'il dépend de la charge de travail de la machine.

Les valeurs dans la colonne "Unix Compil" ont été obtenues sur Unix, avec la même commande "time cc <nom de programme>", les mêmes valeurs ont été retenues comme temps CPU.

Les valeurs dans la colonne "VMS Exec" et "VMS Compil" ont été obtenues sur VMS, avec la commande "show status". Cette commande donne entre autres, le temps CPU depuis le début de la session. Par différence des temps CPU, après une exécution, on peut calculer le temps nécessaire à l'exéctuion d'un programme et le temps nécessaire à la compilation.

Les valeurs dans la colonne "Unix C" et "Unix EXE" ont été obtenues sur Unix, avec la commande "ls -la". Elle renvoie la taille des fichiers sources et exécutables en bytes.

Pour obtenir les valeurs dans la colonne "VMS C" et "VMS EXE", la commande "dir/size" a été employée. Elle renvoie la taille des différents fichiers en bloc de 1/2 KBytes.

3.2 Les tableaux avec les résultats

3.2.1 Les temps moyens

Tableau récapitulatif des moyennes				
Nom	Unix C	Unix Exe	VMS C	VMS Exe
FPRINT	26,62	3,96	4,56	28,19
FTEST	,00	3,98	,50	28,30
WPRINT	25,04	3,76	4,62	29,23
WTEST	,00	3,58	,00	28,41
WHILE	8,16	3,92	4,91	29,50
FOR	8,32	4,02	4,25	28,07
REGISTRE	6,32	3,98	4,24	28,05
ECRIRE	1,94	5,18	10,65	30,67
LIRE	1,56	4,66	7,42	29,75
APPEL	37,30	3,72	26,80	28,31
TESTIF	10,10	3,62	6,83	28,34
PLACE	308	4,08	3,89	29,36
FONCTION	,74	4,14	,66	28,57

3.2.2 Les tailles des fichiers sources et exécutables

Tableau récapitulatif des tailles				
Nom	Unix C	Unix Exe	VMS C	VMS Exe
FPRINT	121	8.192	512	38.912
FTEST	65	5.120	512	38.400
WPRINT	152	8.192	512	38.912
WTEST	58	5.120	512	38.400
WHILE	63	5.120	512	38.400
FOR	57	5.120	512	38.400
REGISTRE	78	5.120	512	38.400
ECRIRE	352	6.144	512	38.912
LIRE	142	7.168	512	38.912
APPEL	102	5.120	512	38.912
TESTIF	75	5.120	512	38.400
PLACE	297	5.120	512	38.912
FONCTION	183	3.656	512	38.400

3.3 Les programmes sources

3.3.1 fprint.c

```
main()
{
     long i;

     for(i=0L; i <= 1000L; i++)
        printf("test 1 - test 2 - test 3 - test 4 - test 5");
}
```

3.3.2 ftest.c

```
main()
{
     long i;

     for(i = 0L; i <= 1000L; i++)
}
```

3.3.3 wprint.c

```
main()
{
     long i;

     i = 0L;
     while(i <= 1000L) {
        printf("test 1 - test 2 - test 3 - test 4 - test 5");
        i++;
        }
}
```

3.3.4 wtest.c

```
main()
{
     long i;

     i = 0L;
     while (i <= 1000L) i++;
}
```

3.3.5 while.c

```
main()
{
     long i;

     i = 0L;
     while(i <= 10000000L)
          i++;
}
```

3.3.6 for.c

```
main()
{
     long i;

     for(i = 0L; i <= 10000000L; i++)
          ;
}
```

3.3.7 registre.c

```
main()
{
     register long i;

     for(i = 0L; i <= 10000000L; i++)
          ;
}
```

3.3.8 ecrire.c

```
#include <stdio.h>
main()
{
     file *fp,*fopen();
     char *mot =    "test_1_test_2_test_3_test_4_test_5";
     int i,j;

     fp = fopen("ecrf", "w")
     j = 0;
     while(j <= 1000) {
          i = 0;
          while (*(mot + i) != '\0') {
               putc (*(mot+i), fp);
               i++;
               }
          j++:
          }
     fclose(fp);
}
```

3.3.9 lire.c

```
#include <stdio.h>
main()
{
     file *fp,*fopen();
     int c;

     fp = fopen("ecrf", "r")
     while((c = getchar(fp) != EOF)
     fclose(fp);
}
```

3.3.10 appel.c

```
main()
{
     long i;

     for(i = 0L; i <= 10000000L; i++)
          incre();
}

incre()
{
     int i;
     i++;
}
```

3.3.11 testif.c

```
main()
{
     long i;

     i = 0L;
     while (i <= 1000000L) {
          if (i >= 0L)
               i++;
          }
}
```

3.3.12 place.c

```
#include <stdio.h>
main()
{
     struct sfault {
          int fnum-ref;
          char fnom-aut[25];
          char fprenom-aut[25];
          };
     long i:
     char *malloc();
     struct sfault *pt;

     i = 0L;
     while (i <= 10000L) {
        pt ((struct sfault *) malloc(sizeof(struct sfault)));
        i++;
        }
}
```

3.3.13 fonction.c

```
main()
{
     long i;
     float res;

     for(i = 0L; i <= 50000L; i++) {
          res = (i + (1/10) * (45 - 18/2 * 10));
     }
}
```